Architecture sécurisée pour les systèmes d'information des avions du futur

Maxime Lastera

▶ **To cite this version:**

Maxime Lastera. Architecture sécurisée pour les systèmes d'information des avions du futur. Systèmes embarqués. INSA de Toulouse, 2012. Français. NNT : . tel-00938782

HAL Id: tel-00938782
https://theses.hal.science/tel-00938782

Submitted on 29 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par:
Institut National des Sciences Appliquées (INSA) de Toulouse

Discipline ou spécialité:
Systèmes Informatiques

Présentée par :
Maxime Lastera

Titre:
Architecture sécurisée pour les systèmes d'information des avions du futur

---

JURY

Rapporteurs :	Michel Cukier	Université du Maryland
	Gilles Muller	INRIA - LIP6
Examinateurs :	Bertrand Leconte	Airbus France
	Laurent Pautet	Télécom ParisTech
Directeurs de Thèse :	Jean Arlat	LAAS-CNRS
	Éric Alata	LAAS-CNRS

École doctorale:
Systèmes (EDSYS)
Unité de recherche:
LAAS-CNRS

Directeur(s) de Thèse:
Jean Arlat & Éric Alata

Rapporteurs:
Michel Cukier & Gilles Muller

A Mes parents,
ma sœur Sophie et ma Claire

Remerciements

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie Messieurs Raja Chatila, Jean-Louis Sanchez et Jean Arlat qui ont assuré succécivement la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire.

Je remercie également Madame Karam Kanoun, Directrice de Recherche CNRS, responsable du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de m'intégrer dans l'equipe et ses activités de recherche.

J'exprime ma très sincère reconnaissance à Messieurs Jean Arlat et Éric Alata, respectivement Directeur de Recherche CNRS et Maître de Conférence, pour m'avoir encadré, soutenu, et encouragé tout au long de cette thèse. Je les remercie pour leurs conseils, leur soutien et leur disponibilité.

J'exprime ma profonde gratitude à Monsieur Laurent Pautet, Professeur à Télécom ParisTech, pour l'honneur qu'il me fait en présidant mon jury de thèse, ainsi qu'à :
- Michel Cukier, Professeur à l'Université du Maryland,
- Gilles Muller, Directeur de Recherche à l'Inria-Lip6 à Paris,
- Bertrand Leconte, Ingénieur à Airbus France à Toulouse,
- Éric Alata, Maître de Conférences à l'Insa à Toulouse,
- Jean Arlat, Directeur de Recherche au LAAS-CNRS à Toulouse,

pour l'honneur qu'ils me font en participant à mon jury.

Je remercie particulièrement Messieurs Michel Cukier et Gilles Muller qui ont accepté la charge d'être rapporteurs.

Comme je l'ai souligné au début, le groupe de recherche TSF est une grande famille, je tiens donc à remercier l'ensemble de ses membres pour l'ambiance qu'ils ont su instaurer.

Merci à l'ensemble des personnes travaillant dans les différents services du LAAS : logistique, magasin, administration... Toutes ces personnes assure le bon fonctionnement du LAAS.

Merci en particulier aux anciens et nouveaux camarades du bureau 10 et 9 : Fernand, Miguel, Rim, Nam, Miruna, Robert, Olivier et Amina, pour leur bonne humeur, leur correction éclairée, les parties de ping-pong constructives. Mention particuliere pour Miruna qui à toujours accepté de m'aider dans la traduction de différents articles. Plus généralement merci aux doctorants du groupe TSF, Anthony, Yann, Yvan, Pierre, Ludovic et Hélène pour les bons moments passés en leur compagnie au laboratoire et en dehors.

Pour finir, je tiens à remercier ma famille, et mes amis qui m'ont soutenu et encouragé, merci à tous.

Enfin, il m'est impossible de ne pas remercier Claire, qui a toujours su me motivé durant mes études.

Table des matières

Introduction

Contexte général

Au cours de ces dernières années, le monde aéronautique s'est illustré par de nombreux faits divers. Citons par exemple le vol QF72 de la compagnie australienne Quantas reliant Singapour à Perth le 3 octobre 2008 [1]. Un problème informatique a causé une chute de l'appareil de 1000 mètres en un peu plus d'une minute, plus précisément, des données incorrectes ont amené les ordinateurs de bord à ordonner un mouvement en piqué. Plus récemment, le 18 juin 2011, un problème de « connectivité sur son réseau », selon les termes de la compagnie aérienne, a obligé United Airlines à annuler 36 vols [2].

Des actes de malveillance sont également rapportés. En juin 2009, au départ de Düsseldorf, un détecteur de fumée d'un A318 est déclaré en panne. Cette alerte n'imposant pas une immobilisation d'urgence, l'appareil est autorisé à rallier sa destination finale : Paris. À l'arrivée, le service de maintenance constate que deux faisceaux électriques avaient été sectionnés [3].

Ces faits divers mettent en évidence le rôle de plus en plus prépondérant joué par les systèmes informatiques au sein des systèmes avioniques et également l'impact de leur communication avec d'autres systèmes de l'environnement avionique. Cela pose de nouvelles problématiques relatives à la sécurité (au sens *security* ou sécurité-immunité) qui viennent s'ajouter aux contraintes de *safety* (ou sécurité-innocuité) déjà connues dans le domaine avionique.

Cette importance croissante de la notion de sécurité-immunité repose sur plusieurs facteurs.Tout d'abord, afin de réduire les coûts de développement, l'industrie aéronautique utilise de plus en plus de composants sur étagère (ou COTS pour Commercial Off-The-Shelf) pour des applications plus ou moins critiques. Ces composants, étant conçus pour être génériques, sont souvent d'une plus grande complexité que ceux destinés à un système particulier. Cette complexité les fragilise vis-à-vis de la sécurité. En effet, il est pratiquement impossible de les vérifier parfaitement, c'est-à-dire de garantir, d'une part, l'absence de bogues, et ,d'autre part, l'absence de fonctions cachées. Par conséquent, des pirates peuvent profiter de cette situation pour réussir à pénétrer dans ces composants et d'en utiliser les ressources associées.

1. Article paru sur lefigaro.fr le 7 octobre 2008
2. Article paru sur lemonde.fr le 18 Juin 2011
3. Article paru sur rmc.fr le 16 juin 2009

L'utilisation de logiciels en source libre tel que le système d'exploitation Android utilisé sur le Boeing 787 Dreamliner afin de fournir les services de divertissement aux passagers procure, du fait de sa disponibilité grand public, davantage d'opportunités pour un attaquant de s'introduire dans le système [4].

Notre travail se situe dans le contexte de l'industrie aéronautique. Plus particulièrement, notre étude est axée sur les communications entre applications de niveaux de criticité différents qui sont amenées à interagir afin de fournir des services innovants. Ces derniers sont mis à disposition des clients toujours plus demandeurs de nouvelles fonctions. Ces interactions posent les problématiques de sécurité-immunité et de sécurité-innocuité. En effet, une application peu critique est développée sans contraintes spécifiques vis-à-vis de la sécurité (immunité et innocuité). L'autorisation d'une interaction entre deux applications de niveaux de criticité différents pourraient corrompre l'application la plus critique et entraîner une défaillance aux conséquences catastrophiques pour le système.

Afin de se prémunir de ces risques de contamination, l'industrie aéronautique autorisait jusqu'à il y a quelques années uniquement des communications unidirectionnelles entre des composants en charge de fonctions de criticités différentes. Ce schéma de communication rappelle le fonctionnement des diodes dans les circuits électroniques.

Cette approche étant trop rigide pour répondre au besoin croissant d'introduction de nouvelles interactions, l'industrie aéronautique a intégré, au fur et à mesure des années, les évolutions des technologies dans ces architectures.

Nous proposons, dans le cadre de notre travail, d'étudier une architecture logicielle qui assure une communication bidirectionnelle entre applications de niveaux de criticité différents. Cette architecture prend évidemment en compte aussi bien les aspects d'immunités que d'innocuités. Pour ce faire, nous utilisons des mécanismes de tolérance aux fautes tels que la diversification fonctionnelle, afin d'augmenter la confiance que nous pouvons apporter à une application peu critique. Cette diversification est obtenue par l'utilisation de la technologie de virtualisation à l'exécution d'une application critique dans un environnement ouvert.

Structure du document

Nous commençons ce manuscrit par un premier chapitre rappelant la terminologie et les définitions du domaine de la sûreté de fonctionnement. Par la suite, nous présentons quelques architectures avioniques embarquées ainsi que leur évolution, étroitement liée aux avancées significatives du domaine du logiciel. Nous mettons enfin l'accent sur la composante sécurité-immunité, qui est le fil conducteur de ce manuscrit, pour l'étude d'une architecture avionique sûre de fonctionnement.

Dans le deuxième chapitre, afin de mieux appréhender notre cas d'étude, nous présentons tout d'abord deux équipements électroniques que sont l'*Electronic Flight Bag* (EFB) utilisé par les pilotes et le portable de maintenance (PMAT pour *Portable Maintenance Access Terminal*) destiné aux opérateurs de maintenance. Ces équipements sont généralement mobiles (par rapport aux systèmes embarqués à bord de l'avion) et ont un niveau

4. Article paru sur h-online.com le 12 juillet 2012

de criticité inférieur à un système avionique. L'évolution des systèmes avioniques nécessite d'autoriser la communication bidirectionnelle entre un tel équipement et un équipement avionique. Dans ce cadre, notre étude se focalise sur l'architecture logicielle et les moyens de détection à mettre en place sur l'EFB ou le PMAT afin de rendre possible ces communications tout en réalisant l'objectif final qui est la protection de l'avion contre des actes de malveillance. Les actions de maintenance, qui représentent notre cas d'étude, seront présentées sous deux scénarios différents afin de montrer leurs évolutions. Par la suite, nous décrivons l'architecture logicielle d'un équipement basée sur le mécanisme de diversification par redondance. Afin d'identifier les attaques, nous présentons un processus de détection basé sur la comparaison des traces d'exécution des entités redondantes. Nous présentons les limites d'un tel processus, ce qui nous amène à nous poser la question suivante : comment repousser les limites du processus de détection ?
La description de l'architecture logicielle permet également de présenter la technologie de virtualisation employée comme outil de diversification. Cette technologie est le fondement de notre architecture et par conséquent nous devons répondre à une question essentielle : l'utilisation de la virtualisation est-elle préjudiciable pour les performances des applications ?

Les deux chapitres suivants apportent des éléments de réponses aux deux questions posées.

En premier lieu, nous proposons dans le troisième chapitre une amélioration du processus de détection d'attaque basée sur un modèle d'exécution de l'application afin d'accroître la couverture de détection. Pour cela, après avoir décrit notre cas d'étude, nous détaillons le principe de notre proposition. Ensuite, nous décrivons le processus de construction du modèle d'exécution de l'application avant de présenter la mise en place de la démarche dans le cycle de vie de l'application. Nous terminons ce chapitre en montrant la complémentarité des mécanismes utilisés afin de détecter une attaque.

En dernier lieu, le quatrième chapitre présente les différents types de virtualisation ainsi que les moyens techniques d'aide à la virtualisation. Nous décrivons ensuite le banc de tests que nous avons utilisé afin d'évaluer les performances d'une solution de virtualisation. Les performances étudiées concernent l'utilisation du processeur, de la mémoire et du réseau.

Le dernier chapitre, plus technique, détaille l'implémentation du démonstrateur réalisé.

Nous concluons ce manuscrit par une synthèse de nos contributions et présentons quelques perspectives de notre travail.

Chapitre I

Sûreté de fonctionnement et architecture embarquée

Introduction

Les systèmes embarqués avioniques comprennent l'ensemble des équipements (électronique, électrique et informatique) permettant de contrôler un avion. Ces équipements, tels que les instruments de navigation ou les systèmes de communication, peuvent, en cas de dysfonctionnement, porter atteinte à la sécurité des passagers. Cette notion de « sécurité » fait partie intégrante du domaine de la sûreté de fonctionnement que nous présentons dans ce chapitre. Dans un premier temps et afin de faciliter la lecture, nous introduirons une partie du vocabulaire, en considérant les concepts et la terminologie introduits dans le « Guide de la Sûreté de Fonctionnement » [Laprie *et al.* 1996] et mis à jour dans [Avizienis *et al.* 2004]. Plus précisément, nous commençons par définir les notions de sûreté de fonctionnement, pour ainsi situer nos travaux par rapport au thème et aux moyens classiques de la sûreté de fonctionnement en général, et de la sécurité informatique en particulier. Dans un second temps, nous présentons une architecture utilisée dans la conception des systèmes embarqués et les normes mises en œuvre pour assurer les propriétés de sécurité.

I.1 Sûreté de fonctionnement : notions et taxinomie

La sûreté de fonctionnement d'un système informatique est, selon [Laprie *et al.* 1996], [Avizienis *et al.* 2004], l'aptitude à délivrer un service avec un niveau de confiance justifiée. Ce service délivré par un système est son comportement tel qu'il est perçu par son ou ses utilisateurs, l'utilisateur étant un autre système, humain ou non. La sûreté de fonctionnement informatique comporte trois principaux volets : les attributs qui la décrivent, les entraves qui nuisent à sa réalisation et les moyens de l'atteindre (*cf.* Figure I.1).

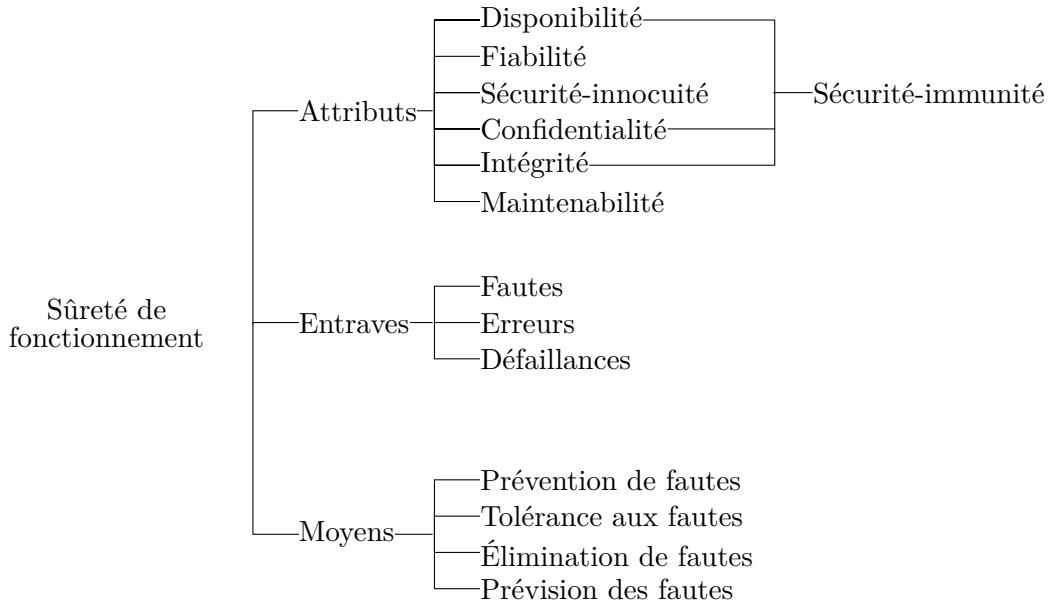

Figure I.1 – Arbre de la sûreté de fonctionnement

I.1.1 Attributs

La sûreté de fonctionnement d'un système peut être perçue selon différentes propriétés. Ces propriétés sont appelées les **attributs** de la sûreté de fonctionnement. Les attributs à considérer dépendent des applications auxquelles le système est destiné. Ces attributs sont au nombre de six :

- La disponibilité : aptitude du système à être prêt à l'utilisation ;
- La fiabilité : continuité du service ;
- La sécurité-innocuité : absence de conséquences catastrophiques pour l'environnement ;
- La confidentialité : absence de divulgations non autorisées de l'information ;
- L'intégrité : absence d'altérations inappropriées de l'information ;
- La maintenabilité : aptitude aux réparations et aux évolutions.

Ces attributs peuvent être combinés pour aboutir à un nouvel attribut. Par exemple, la confidentialité, l'intégrité et de la disponibilité se combinent pour aboutir à la sécurité-immunité.

I.1.2 Entraves

Une entrave à la sûreté de fonctionnement est une circonstance indésirable, mais non inattendue. Elle est la cause ou le résultat de la non-sûreté de fonctionnement. Nous en distinguons trois sortes : les défaillances, les erreurs et les fautes.

Une **défaillance** survient lorsque le service délivré par le système s'éloigne de l'accomplissement de la fonction du système. Néanmoins, un système ne défaille pas toujours de la même manière, ce qui conduit à définir la notion de mode de défaillance qui peut être caractérisé selon trois points de vues : domaine de défaillance, conséquence des défail-

lances sur l'environnement du système et perception des défaillances par les utilisateurs du système.

Le domaine de défaillance est divisé en deux catégories :

- Les défaillances en valeur : la valeur du service délivré ne permet plus l'accomplissement de la fonction du système ;
- Les défaillances temporelles : le service n'étant pas délivré dans le temps imparti, la fonction du système ne peut pas être accomplie.

Les conséquences des défaillances conduisent à distinguer :

- Les défaillances bénignes, dont les conséquences sont du même ordre de grandeur que le bénéfice procuré par le service délivré en l'absence de défaillance ;
- Les défaillances catastrophiques, dont les conséquences sont infiniment différentes du bénéfice procuré par le service délivré en l'absence de défaillance.

Lorsqu'un système a plusieurs utilisateurs, la perception des défaillances conduit à distinguer :

- Les défaillances cohérentes : tous les utilisateurs du système ont la même perception du service délivré ;
- Les défaillances incohérentes : les utilisateurs du système peuvent avoir des perceptions différentes des défaillances ; les défaillances incohérentes sont souvent qualifiées de défaillances byzantines.

La gravité de la défaillance d'un composant définit son niveau de criticité, et par conséquent fixe les exigences de développement qui sont requises.

Les **erreurs** sont le deuxième type d'entraves à la sûreté de fonctionnement. Une erreur est la partie de l'état du système susceptible d'entraîner une **défaillance** : une erreur affectant le service est une indication qu'une défaillance survient ou est survenue. Une erreur peut être latente ou détectée ; une erreur est latente tant qu'elle n'a pas été reconnue en tant que telle ; une erreur est détectée par un algorithme ou un mécanisme de détection. Une erreur peut disparaître sans être détectée.

Une **faute** est la cause adjugée ou supposée d'une erreur. Les fautes sont de nature extrêmement diverses et peuvent être classées selon cinq points de vue : leur cause phénoménologique (fautes physiques ou fautes dues à l'homme), leur nature (fautes accidentelles ou fautes délibérées), leur phase de création ou d'occurrence (fautes de développement ou fautes opérationnelles), leur situation par rapport aux frontières du système (fautes internes ou externes) et leur persistance (fautes permanentes ou fautes temporaires).

Une faute est active lorsqu'elle produit une erreur. Une faute active est soit une faute interne, qui était préalablement dormante et qui a été activée par le processus de traitement, soit une faute externe. Une faute interne peut changer de nature, et ainsi passer d'un état dormant à actif et inversement.

Par propagation, une erreur crée de nouvelles erreurs. Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système. Cette défaillance peut alors apparaître comme une faute du point de vue d'un autre composant. On obtient ainsi la chaîne fondamentale suivante :

$$\ldots \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow \text{erreur} \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow \ldots$$

Les flèches dans cette chaîne expriment la relation de causalité entre fautes, erreurs et défaillances. Elles ne doivent pas être interprétées au sens strict ; par propagation, plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne. Pour minimiser l'impact de ces entraves sur les attributs retenus d'un système, la sûreté de fonctionnement dispose de moyens.

I.1.3 Moyens

Ces moyens sont les méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement de la fonction du système. Ces moyens peuvent être utilisés simultanément lors de la phase de conception et développement d'un système sûr de fonctionnement. Ils sont classés suivant l'objectif visé :

- La prévention : empêcher l'occurrence ou l'introduction de fautes.
- La tolérance : fournir un service qui remplit la fonction du système en dépit des fautes.
- L'élimination : réduire la présence (nombre, sévérité) des fautes.
- La prévision : estimer la présence, la création et les conséquences des fautes.

Ces moyens sont complémentaires, dépendants et doivent être utilisés de façon combinée. En effet, en dépit de la prévention des fautes grâce à des méthodes de conception et à des règles de construction rigoureuses, des erreurs surviennent, résultant en des fautes. D'où le rôle de l'élimination des fautes : lorsqu'une erreur est révélée durant la vérification, un diagnostic est entrepris afin de déterminer la, ou les fautes causes de l'erreur, en vue de les éliminer. Cette élimination étant elle-même imparfaite, il est également nécessaire d'effectuer de la prévision de fautes. Et, bien entendu, la dépendance croissante que nous avons dans les systèmes informatiques conduit à mettre en œuvre de la tolérance aux fautes.

Dans le cadre de nos travaux, nous nous intéressons essentiellement à la tolérance aux fautes. Nous partons donc de l'hypothèse que des fautes peuvent toujours exister et se manifester, et faisons en sorte que les architectures que nous étudions soient robustes vis-à-vis ces fautes. Nous proposons de détailler dans ce qui suit la tolérance aux fautes, et plus particulièrement la notion de redondance, qui est un principe essentiel de la tolérance aux fautes. En effet, pour décider de la validité d'un état ou non (et par conséquent détecter une erreur), il faut disposer d'un critère permettant d'effectuer ce processus de décision. Le critère peut prendre la forme d'un autre état représentant la même information, ou la forme d'une valeur de référence à comparer avec la valeur observée. Cependant, quelle que soit la nature de ce critère, ce dernier doit contenir une information permettant de valider l'état en question, et cette information est toujours une forme de redondance de l'état contrôlé.

I.2 Tolérance aux fautes

Comme nous l'avons précédemment indiqué, la tolérance aux fautes est le moyen de la sûreté de fonctionnement qui cherche à maintenir un service correct délivré par le système malgré la présence de fautes.

La tolérance aux fautes est mise en œuvre par le traitement d'erreur et par le traitement de faute [Lee & Anderson 1990]. Le traitement d'erreur est destiné à éliminer les erreurs, de préférence avant qu'une défaillance ne survienne. Le traitement de faute est destiné à éviter qu'une ou des fautes ne soient activées de nouveau. Dans le cadre de nos travaux, nous nous intéressons au traitement d'erreur. Ce traitement s'articule autour de mécanismes élémentaires de la tolérance aux fautes. Tout d'abord nous trouvons **les mécanismes de détection d'erreur**, qui permettent de détecter la présence d'une erreur après l'activation d'une faute. Ensuite viennent **les mécanismes de recouvrement d'erreur** qui transforment l'état d'un système en substituant un état exempt d'erreur à un état erroné.
Nous proposons de détailler ces mécanismes dans la suite de cette section.

I.2.1 Mécanismes de détection

Cette section présente les principaux mécanismes de détection d'erreur les plus à même d'être utilisés dans le cadre de nos travaux.

- L'utilisation de **codes détecteurs voire correcteurs d'erreurs** [Peterson & Weldon 1972] consiste à transformer les données (les coder) en ajoutant de la redondance de manière à ce qu'une erreur à la réception (décodage) soit directement détectée. La capacité de détection d'erreurs est liée à la complexité du code.
- La **duplication** et **comparaison** utilise au moins deux unités redondantes qui doivent être indépendantes vis-à-vis des fautes que l'on souhaite tolérer : typiquement, la redondance des composants matériels pour les fautes physiques, et la diversification pour des fautes de conception.
- Les **chiens de garde** (*watchdogs*) [Namjoo 1982] sont typiquement utilisés pour contrôler le temps d'exécution d'une tâche ou le temps de réponse d'un périphérique donné. Le chien de garde s'assure que ce temps ne dépasse pas une valeur seuil précédemment définie (*time-out*). Cette technique a pour avantage son faible coût.
- Les **contrôles de vraisemblance** ont pour objectif de détecter des erreurs en valeur aberrantes pour le système. Ils peuvent être mis en œuvre soit par du matériel pour détecter par exemple des accès à des adresses mémoire interdites ou inexistantes, soit par du logiciel pour vérifier la conformité des entrées, des sorties ou des variables internes du système.
- L'utilisation de la **programmation défensive** [Rabéjac 1995] consiste à ajouter des contrôles sur les entrées et les sorties des modules sous la forme d'assertions, servant à détecter s'il y a erreur et éventuellement déclencher un traitement d'exception. Le traitement d'exception constitue alors une manière d'éviter que la défaillance d'une tâche n'entraîne la défaillance de tout le système.
- La **vérification de l'exécution par rapport à un modèle** consiste à vérifier, durant l'exécution d'une application, que toutes les étapes du modèle théorique de l'application sont exécutées. Cette vérification est assurée en exécutant un programme qui parcourt le graphe d'état traduisant le modèle en même temps que l'exécution de l'application. Chaque résultat intermédiaire correspond à un état dans le graphe du modèle et l'on vérifie ainsi la cohérence entre l'exécution du programme et le modèle [Ayache *et al.* 1979].

I.2.2 Mécanismes de recouvrement

Nous citons tout d'abord quatre principaux mécanismes de recouvrement.

1. La **reprise** consiste à ramener le système dans un état sain préalablement sauvegardé. La sauvegarde doit se faire de manière pèriodique, ce qui nécessite un espace de stockage conséquent. De plus, le surcoût temporel dû au rétablissement du système peut ne pas être négligeable.
2. La **poursuite** a pour nature la recherche d'un nouvel état acceptable pour le système à partir duquel celui-ci pourra fonctionner (éventuellement en mode dégradé). Ainsi, la poursuite consiste souvent à réinitialiser le système en essayant d'acquérir de nouvelles données depuis l'environnement externe d'exécution.
3. La **détection-compensation** permet, lors de la détection d'une erreur, de reconstituer un état exempt d'erreur, afin d'être en mesure de poursuivre le traitement, ce qui nécessite généralement une redondance forte. Un exemple classique est l'utilisation de composants autotestables : en cas de défaillance de l'un d'entre eux, il est déconnecté et le traitement se poursuit sans interruption sur les autres. La compensation repose alors sur la commutation d'un composant à un autre. Le système de gestion des commandes de vol des Airbus A320 est basé sur cette technique [Traverse *et al.* 2004].
4. Le **masquage**, contrairement à la compensation qui fait suite à la détection d'une erreur, est exécuté de façon systèmatique, en présence ou non d'erreur. L'utilisation d'un algorithme de vote portant sur les résultats produits par un nombre suffisant de répliques constitue la principale technique parmi les différentes options possibles pour implémenter la prise de décision dans le cas du masquage. [Lorczak *et al.* 1989]

Une spécialisation des ces mécanismes dans le cas des fautes de conception donne lieux aux mécanismes suivants.

- Les **blocs de recouvrement** [Randell 1975] basés sur le principe de la reprise, utilisent plusieurs variantes exécutées séquentiellement jusqu'à obtenir un résultat acceptable. Le décideur fait alors appel à la variante suivante s'il considère que le résultat fourni par une première variante est invalide. Dans le cas où aucune variante ne fournit de résultat valide, une alerte est levée pour traiter l'erreur à un niveau englobant.
- La **programmation en N-versions** [Avizienis 1985] se base sur le même principe que les blocs de recouvrement, sauf que les variantes ne sont pas exécutées de manière optionnelle mais systématiquement, généralement de façon simultanée. Dans ce cas, les résultats des variantes peuvent être différents, même en l'absence d'erreur, en raison de la diversification. Il faut donc traiter les résultats des variantes (par exemple au moyen d'un vote) pour produire un résultat valide cohérent entre les variantes. Cette technique offre une bonne disponibilité du système, même en cas d'activation de fautes, puisque le surcoût en temps peut être négligeable ; cependant, elle crée un surcoût de traitement permanent et régulier (N exécutions au lieu d'une). Ainsi, une programmation N-versions utilise, comme son nom l'indique, plusieurs versions que l'on évalue pour obtenir un résultat correct. La cohérence entre ces différentes versions présente alors le critère de validité précédemment mentionné. Les blocs de recouvrement minimisent cette surcharge d'exécution, mais en cas d'activation de faute, une surcharge temporelle doit être prise en compte.
- La **programmation N-autotestable** [Laprie *et al.* 1990] met en œuvre des composants autotestables exécutés en parallèle. Chaque composant autotestable peut

être un bloc de recouvrement ou même un composant N-versions. Comme pour la programmation en N-versions, les composants corrects ne fournissent pas nécessairement un résultat identique. Il faut donc traiter les résultats des variantes pour produire un résultat valide cohérent entre les variantes. Cependant, ce traitement des résultats est simplifié par la propriété d'autotestabilité des composants.

I.2.3 Tolérance aux fautes malveillantes

Les mécanismes de tolérance aux fautes présentés dans le paragraphe précédent ont été conçus pour être génériques et applicables à différents contextes. Une adaptation au contexte particulier de la sécurité informatique et aux malveillances a fait l'objet de travaux effectués dans le cadre du projet MAFTIA [MAFTIA 2003].

Le projet MAFTIA a examiné une approche pour tolérer tant les fautes accidentelles que des attaques malveillantes dans des systèmes distribués à grande échelle, leur permettant ainsi de rester opérationnels en cas d'attaque, sans exiger une longue durée d'adaptation ou d'intervention humaine potentiellement prédisposée aux erreurs. Plus particulièrement, une terminologie relative aux malveillances a été introduite dans le cadre de ce projet. Nous en reprenons quelques définitions.

Les fautes malveillantes se déclinent en deux classes principales : les logiques malignes et les intrusions. Les logiques malignes sont des parties du système conçues pour provoquer des dégâts (bombes logiques) ou pour faciliter des intrusions futures (vulnérabilités créées volontairement). Elles peuvent être introduites dès la conception du système (par un concepteur malveillant), ou en phase opérationnelle (par l'installation d'un logiciel contenant un cheval de Troie ou par une intrusion). La définition d'une intrusion est étroitement liée aux notions d'attaque et de vulnérabilité :

- Une **attaque** est une faute d'interaction malveillante, par laquelle un attaquant viole délibérément une ou plusieurs propriétés de sécurité. Il s'agit d'une faute externe créée avec l'intention de nuire (vers, virus, etc.).
- Une **vulnérabilité** est une faute créée pendant la phase de développement du système, ou lors de son utilisation, qui pourrait être exploitée par une attaque pour créer une intrusion.
- Une **intrusion** est une faute malveillante, résultant d'une attaque qui a réussi à exploiter une vulnérabilité.

Du point de vue des fautes malveillantes, un système tolérant aux intrusions est un système capable de s'auto-diagnostiquer, se réparer et se reconfigurer tout en continuant à fournir un service acceptable aux utilisateurs légitimes pendant une attaque [Deswarte *et al.* 1991].

Les méthodes de détection des intrusions visent à détecter des atteintes à la politique de sécurité d'un système. Deux démarches peuvent s'appliquer : une dite comportementale et l'autre dite par scénario.

La démarche comportementale se base sur l'hypothèse qu'un comportement malveillant entraîne une activité inhabituelle sur le système. Elle compare le comportement observé du système et une référence de comportement normal (on parle alors de détection d'anomalie quand le comportement observé s'éloigne de la référence). [Deswarte & Powell 2006]

Quant à la démarche par scénarios, elle fonctionne avec une base de signatures de scénarios anormaux. Elle détecte une atteinte à la politique de sécurité lorsqu'une séquence d'informations liée à un comportement possède une signature référencée dans la base de signatures de scénarios anormaux. Cette base doit être constamment mise à jour, sous peine de ne pas détecter les attaques non connues. De manière générale, les mécanismes de détection d'intrusions soulèvent un nombre important de fausses alertes mais également certaines violations de la politique de sécurité ne sont pas détectées. Nous pouvons également citer l'**authentification** (des utilisateurs) et l'**autorisation** (vérification des droits d'accès) qui sont les moyens de contrôle d'accès classiques en informatique. Ils constituent un moyen pour détecter les malveillances (en plus d'empêcher les intrusions).

La tolérance aux intrusions peut être appliquée avec différentes techniques de sécurité parmi lesquelles, le **chiffrement** qui consiste à assurer la confidentialité des données échangées par l'utilisation d'une clé secrète initialement partagée par les correspondants. La **réplication** assure la disponibilité des données en utilisant plusieurs source de données redondantes. Le **brouillage des données** utilise comme le chiffrement une clé secrète initialement partagée par les correspondants afin d'effectuer une permutation des données de manière à les rendre illisibles.

Ces techniques, bien que tolérant les intrusions, ne prennent pas en compte tous les attributs de sécurité de l'information, à savoir, la confidentialité, l'intégrité et la disponibilité. Il existe une technique plus évoluée qui tolère les intrusions tout en assurant la sécurité de l'information échangée. La technique de fragmentation, redondance et dissémination.

Cette technique permet de tolérer les intrusions tout en protégeant la sécurité des informations confidentielles. Pour ce faire, elle découpe l'information en fragments, qui ne peuvent pas fournir, à eux seuls, des informations significatives. Ces fragments sont ensuite dupliqués et dispersés sur différents sites. De la redondance est ajoutée aux fragments (par réplication ou par codage) pour permettre de détecter la modification ou la destruction de fragments, et de reconstituer l'information même si des fragments ont été modifiés ou détruits [Fabre *et al.* 1996].

La redondance avec diversification s'appuie sur un constat simple : une attaque qui cible une vulnérabilité d'un système fonctionnant sur une architecture matérielle ou logicielle particulière a une infime probabilité de fonctionner sur un autre système utilisant une autre architecture. Le principe de cette technique est donc d'utiliser plusieurs sous-systèmes différents sur des architectures différentes et fournissant le même service, avec un mécanisme de vote majoritaire pour former le système global. De la sorte, une attaque susceptible d'engendrer des dégâts sur un des sous-systèmes sera inefficace sur les autres. Le système global sera toujours disponible sous réserve, au moins, que la majorité des sous-systèmes soient aussi disponibles. [Deswarte *et al.* 1991]

Dans la suite de cette section, nous présentons comment les architectures embarquées pour l'avionique intègrent des mécanismes de sûreté de fonctionnement et plus particulièrement de tolérance aux fautes.

I.3 Système avionique

Les systèmes avioniques sont soumis à de nombreuses exigences spécifiées par les autorités de certification [1].

Des recommandations ou des normes internationales ont été émises par ces autorités pour harmoniser les exigences. Nous débutons cette section par une brève présentation des normes et standards liés à la sécurité-innocuité qui encadrent la conception des systèmes avioniques. Nous présenterons ensuite les contraintes de sécurité-immunité liées au déploiement de tels systèmes.

I.3.1 Normes et standards pour l'avionique embarquée

Considération de sécurité-innocuité

L'utilisation croissante du logiciel dans les systèmes et équipements embarqués mis en place dans les avions au début des années 1980 a entraîné la nécessité pour l'industrie de définir des guides pour satisfaire les exigences de navigabilité. La norme DO-178C/ED-12C, faisant suite à la normes DO-178B/ED-12B « Considérations logicielles dans les systèmes aéroportés et de certification du matériel », a été écrite pour satisfaire ce besoin [DO 178C 2011].

La démarche générale repose sur un processus d'analyse de la sécurité-innocuité qui consiste à examiner les fonctions et l'architecture des systèmes avioniques et à identifier les gravités de leurs défaillances. Cinq catégories de gravités sont définies (*cf.* [DO 178C 2011]) :

1. Catastrophique : défaillance qui entrainerait de nombreuses pertes de vies humaines, la destruction probable de l'avion ;
2. Dangereux : défaillance qui réduirait la capacité de l'avion ou l'aptitude de l'équipage à faire face à des conditions d'exploitation défavorables dans la mesure où il y aurait :
    - Une réduction importante des marges de sécurité ou des capacités fonctionnelles ;
    - Une détresse physique ou une charge de travail excessive, ne permettant pas á l'équipage d'éffectuer ses tâches avec précision ;
    - Des blessures graves ou mortelles sur un faible nombre d'occupants autres que l'équipage.
3. Majeure : défaillance qui réduirait la capacité de l'avion ou l'aptitude de l'équipage à gérér des conditions d'exploitation défavorables dans la mesure où il serait, par exemple, question d'une réduction significative des marges de sécurité ou des capacités fonctionnelles, une augmentation significative de la charge de travail de l'équipage dans des conditions qui portent atteinte à son efficacité, son confort ou son intégrité physique.

1. Ces autorités sont spécifiques aux pays du constructeur. Aux États-Unis, l'autorité compétente est l'administration fédérale d'aviation FAA (`Federal Aviation Administration`), pour l'Europe, il s'agit de l'agence européenne de la sécurité aérienne EASA (`European Aviation Safety Agency`).

4. Mineure : défaillance qui ne réduirait pas de manière significative la sécurité des avions, et qui implique des actions de l'équipage qui sont bien en deçà de leurs capacités. Cette défaillance mineure peut inclure, par exemple, une légère réduction des marges de sécurité ou des capacités fonctionnelles, une légère augmentation de la charge de travail de l'équipage, tels que les changements de plan de vol de routine ;
5. Sans effet : défaillance qui n'a aucun effet sur la sécurité-innocuité, par exemple une défaillance qui n'affecte pas les capacités opérationnelles de l'avion et qui n'augmente pas la charge de travail de l'équipage.

Ces catégories de gravités sont à mettre en relation avec les niveaux logiciels, communément nommés niveau DAL (Development Assurance Level) ou IDAL (Item Development Assurance Level). L'attribution d'un niveau de DAL à une fonction avionique est réalisée après une analyse de risques du système, visant à rendre l'appareil sûr de fonctionnement. Cinq niveaux DAL sont définis :

- Niveau A : Logiciel dont le comportement anormal pourrait contribuer à une défaillance de la fonction du système résultant dans un état de défaillance catastrophique de l'avion.
- Niveau B : Logiciel dont le comportement anormal pourrait contribuer à une défaillance de la fonction du système résultant en une condition de défaillance dangereuse pour l'avion.
- Niveau C : Logiciel dont le comportement anormal pourrait contribuer à une défaillance de la fonction du système résultant en une condition d'échec majeur pour l'avion.
- Niveau D : Logiciel dont le comportement anormal pourrait contribuer à une défaillance de la fonction du système résultant en une condition de défaillance mineure pour l'avion.
- Niveau E : Logiciel dont le comportement anormal pourrait contribuer à une défaillance de la fonction du système, mais qui est sans effet sur la capacité opérationnelle de l'avion ou la charge de travail du pilote.

Cette classification détermine le niveau d'assurance que le système doit satisfaire. Ainsi, les logiciels les plus critiques sont développés au niveau DAL-A alors que les logiciels qui n'ont aucun impact sur la sécurité de l'appareil sont développés au niveau DAL-E (correspondant généralement au niveau des applications existantes dans le monde ouvert).

Comme nous l'avons précédemment indiqué, la norme DO-178 est consacrée au processus de développement des logiciels embarqués. Elle est basée sur une approche orientée processus. On distingue trois types de processus dans le cycle de vie du logiciel :

- Le processus de planification qui coordonne le processus de développement et les processus intégraux ;
- Le processus de développement, qui comprend la spécification, la conception, le codage et l'intégration ;
- Le processus intégral, qui comprend la vérification, la gestion de configuration, l'assurance qualité et la coordination pour la certification.

Pour chaque processus et chaque niveau d'assurance, les objectifs sont définis et une description des données du cycle de vie permettant de démontrer que les objectifs sont satisfaits, est fournie. La norme n'impose aucune méthode pour atteindre les objectifs fixés. C'est à la charge du postulant à la certification (donc les constructeurs avioniques) d'apporter les preuves que les méthodes employées permettent de répondre aux objectifs. De même, bien que la norme ne l'impose pas, elle recommande d'utiliser des stratégies

de tolérance aux fautes qui peuvent être employées pour la détection et le recouvrement d'erreurs.

Comme nous l'avons vu à la Section I.2, la redondance est un des mécanismes les plus employés pour tolérer les fautes aussi bien accidentelles que malveillantes. Ce principe de redondance est largement employé dans le domaine avionique.

La redondance logicielle diversifiée a été identifiée dans la norme DO-178C comme une technique permettant de limiter l'impact des fautes ou de pouvoir les détecter tout en assurant le fonctionnement du système en présence de fautes. Nous proposons de présenter les considérations liées à une telle diversification logicielle, sachant que ces considérations guideront les analyses de fautes menées dans le cadre de l'étude présentée dans ce manuscrit.

L'utilisation d'une redondance mettant en œuvre la diversification (*cf.* Section I.2 pour la terminologie utilisée) permet d'augmenter le niveau de confiance que l'on peut avoir dans le résultat fourni par l'ensemble des logiciels diversifiés. Cependant, l'ajout de mécanismes de diversification doit s'accompagner d'un processus de vérification qui doit en particulier prouver qu'un tel ajout n'a pas d'effet de bord sur les propriétés de sécurité-innocuité du système. La diversification logicielle est ainsi assurée en combinant les règles de diversification suivantes :

- Le code source est implémenté en utilisant deux ou plusieurs langages de programmation différents ;
- Le code exécutable est généré en utilisant un ou plusieurs compilateurs différents ;
- Chaque version de code exécutable est exécutée sur un processeur différent. Dans le cas où un seul processeur serait utilisé, il faut s'assurer du bon fonctionnement des mécanismes d'isolation entre les versions exécutées ;
- L'expression des besoins logiciels et la conception du logiciel, et/ou du code source sont effectués par deux ou plusieurs équipes de développement dont les interactions sont controlées ;
- L'expression des besoins, la phase de conception et d'implantation sont réalisées sur plus de deux environnements de développement. De plus, il est possible que chaque version mise en œuvre soit vérifiée sur des environnements de test différents ;
- Le code exécutable est lié puis chargé en utilisant successivement deux ou plusieurs éditeurs de liens et chargeurs ;
- L'expression des besoins logiciels, la conception et l'implémentation de chaque version doivent être réalisées en utilisant respectivement des standards de besoins logiciels (Software Requirements Standards), standards de conception et standards de code source différents.

Ainsi ces différentes règles peuvent être combinées pour implanter des applications logicielles diversifiées. Dans ce cas, le processus de vérification dépend de la nature de la règle (ou des règles) choisie pour la diversification, en particulier en fonction de la diversification introduite au niveau matériel et/ou logiciel. Cependant, il faut également montrer que les versions sont compatibles entre elles, vis-à-vis des spécifications (et ce pour un fonctionnement normal ou en présence de fautes). De plus, il faut montrer que la solution diversifiée est au moins aussi performante qu'un système non diversifié par rapport à la détection d'erreurs.

Considération de sécurité-immunité

Les systèmes avioniques sont développés en respectant la classification de la famille de norme DO-178 en affectant à chaque composant logiciel un niveau de criticité. Cette même vision de niveaux de criticité est utilisée dans [ARINC 811 2005] pour distinguer différents domaines de sécurité au sein de l'avion.

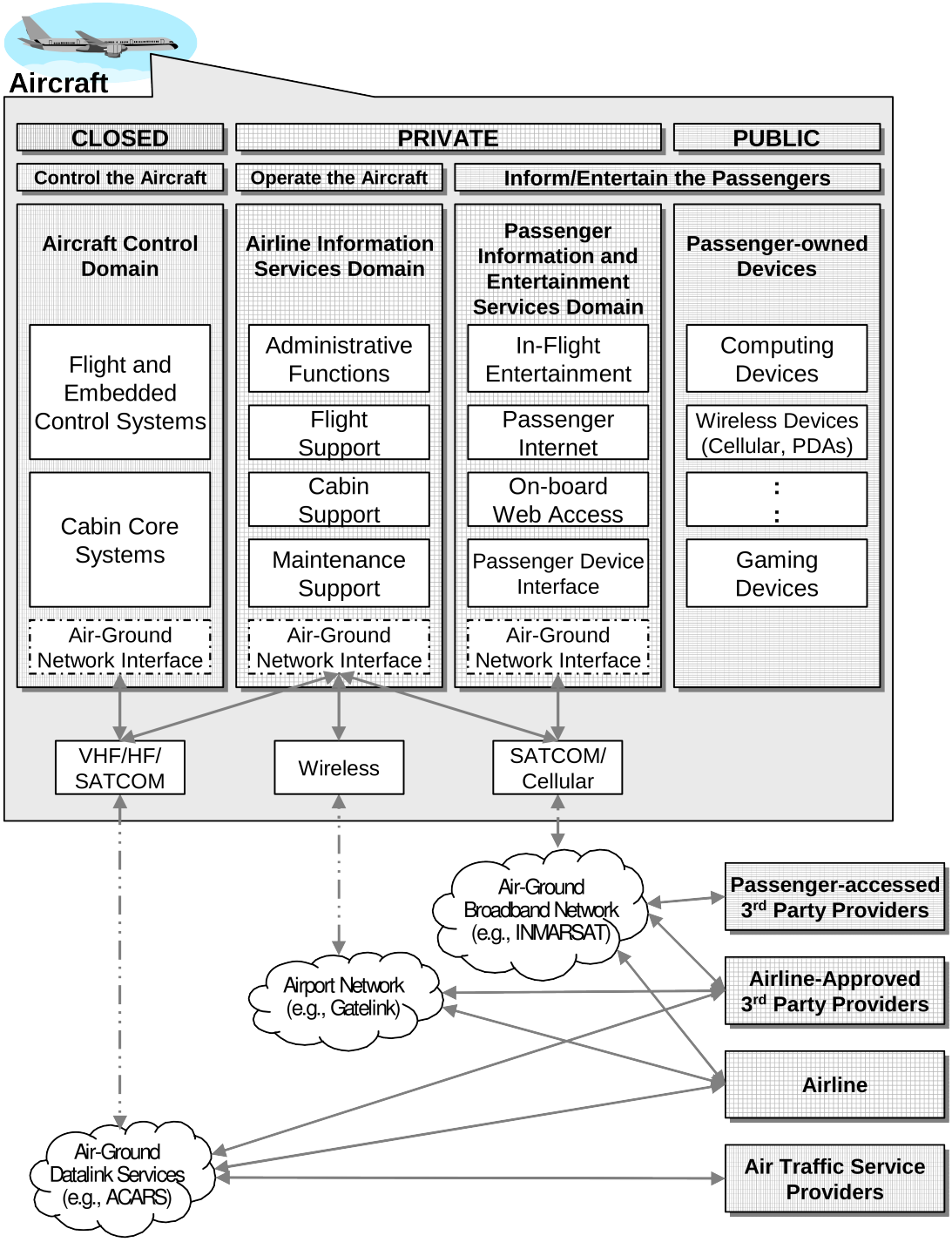

Figure I.2 – Domaines avioniques et leurs interconnexions (ARINC 811)

Comme le montre la Figure I.2, quatre domaines sont définis en fonction de différents

critères : sécurité (existence de contrôle ou non), responsabilité (constructeur, compagnie aérienne, passager), opération aérienne (secrète, privée, publique) et rôle (commande de l'avion, exploitation avionique, information et divertissement des passagers). Nous présentons dans ce qui suit les différents domaines identifiés dans la Figure I.2 et détaillés dans le rapport ARINC 811 [ARINC 811 2005], sans pour autant détailler les spécifications internes de chaque domaine :

- Le domaine de commande de l'appareil (*Aircraft Control Domain*) est le domaine le plus critique dans le monde avionique. Il contient les applications de contrôle et de commande de l'appareil. Ces applications sont généralement installées sur des calculateurs qui récupèrent les commandes du pilote, les transforment (en fonction des lois de pilotage et des données environnementales, par exemple) en données numériques, et les communiquent (via un réseau dédié) aux différents actionneurs de l'appareil.
- Le domaine de services d'information de la compagnie (*Airline Information Services Domain*) contient les différents supports relatifs au vol, la cabine, la maintenance (*cf.* Figure I.2). Ce domaine est moins critique que le précédent, mais il reste tout de même d'un niveau de criticité élevé. En effet, les informations contenues dans ce domaine sont critiques pour l'exploitation de l'appareil (surtout par rapport à la maintenance) par les compagnies aériennes.
- Le domaine de services d'information et de divertissement des passagers (*Passenger Information and Entertainment Services Domain*) est en charge de la communication avec les passagers. Il comprend la gestion des écrans de divertissement (*In Flight Entertainment* : IFE) ainsi que l'interface de connexion des périphériques du passager. Ce domaine est fortement lié à l'image de marque de la compagnie aérienne auprès des passagers. Aussi, les compagnies accordent une grande importance aux équipements de divertissement des passagers, certaines allant même jusqu'à refuser l'autorisation de décollage d'un appareil si un terminal de divertissement (IFE) est en panne.
- Le domaine des équipements des passagers (*Passenger-owned Devices*) est réservé à tous les équipements électroniques des passagers (ordinateurs portables, smartphone, consoles de jeu). Dans certains avions récents, ces équipements sont connectables, via des interfaces spécifiques (présentes dans le domaine précédent), à un réseau que la compagnie aérienne peut configurer en fonction de sa stratégie commerciale (par exemple proposer une connexion Internet aux passagers).

La communication inter-domaine est complètement contrôlée et ne permet pas d'échange de flux d'un niveau peu critique (par exemple le domaine des équipements des passagers) à un niveau plus critique (par exemple le domaine de commande de l'avion). Une telle séparation assure ainsi la sécurité-immunité des applications critiques vis-à-vis des attaques, en supposant que toute information arrivant à ces applications critiques est de confiance. Remarquons que chaque domaine a également la possibilité de se connecter à des réseaux externes bien identifiés. Ces communications sont appelées communications bord-sol.

Sécurité-immunité pour communications bord-sol

Les moyens de communications bord-sol présentés dans la Figure I.3 sont basés historiquement sur des liaisons hertziennes directes (par exemple des communications afin d'obtenir des informations relatives à la position de l'avion) aux instruments de bord

(VHF, UHF). De nos jours, ces communications sont plus diversifiées de par leur utilisation (satellite, WiFi).

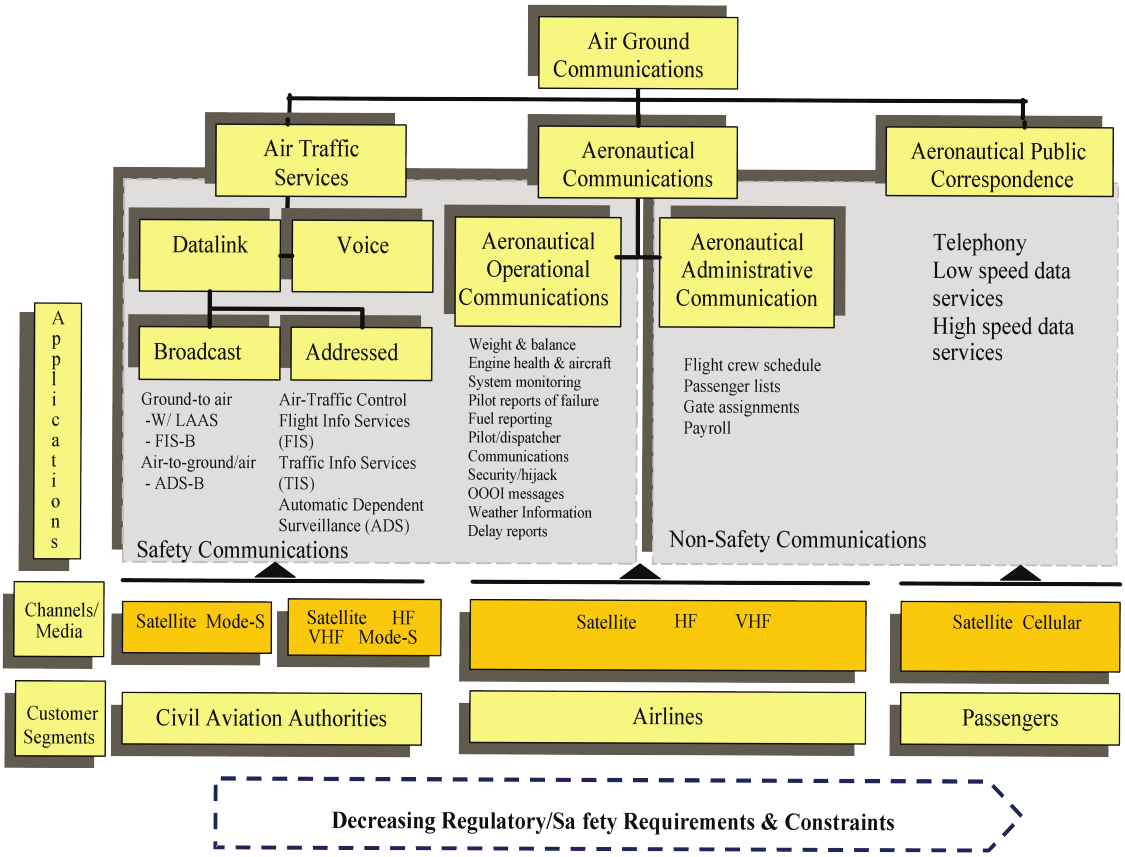

Figure I.3 – Communications bord-sol (ARINC 811)

Ces communications bord-sol sont séparées selon des exigences de sécurité-innocuité. Nous retrouvons ainsi une correspondance avec les domaines avioniques présentés par la Figure I.2. En effet les communications bord-sol sont réglementées selon qu'elles soient issues du trafic aérien (*Airframer*), des compagnies aériennes (*Airline*) ou des passagers (*Passenger*). Ainsi quatre catégories de communication sont identifiées : Les communications de contrôle du traffic aérien (ATC) sont prises en charge par l'Air Trafic Service (ATS), les communications issue des compagnies aérienne (AC) sont décomposées en communication opérationnelles (AOC) et administratives (AAC). Les communications effectuées par les passagers sont prises en charge par l'Aeronautical Public Correspondance (APC).

Ces communications n'ont pas toutes les mêmes exigences en terme de sécurité-innocuité et ne sont donc pas réglementées de la même manière.

La Figure I.4 présente les différents moyens de communications regroupés en quatre domaines :

- *Air Traffic Control* (ATC) ;
- *Aeronautical Operational Control* (AOC) ;

- *Aeronautical Administrative Communication* (AAC) ;
- *Aeronautical Passenger Correspondence* (APC).

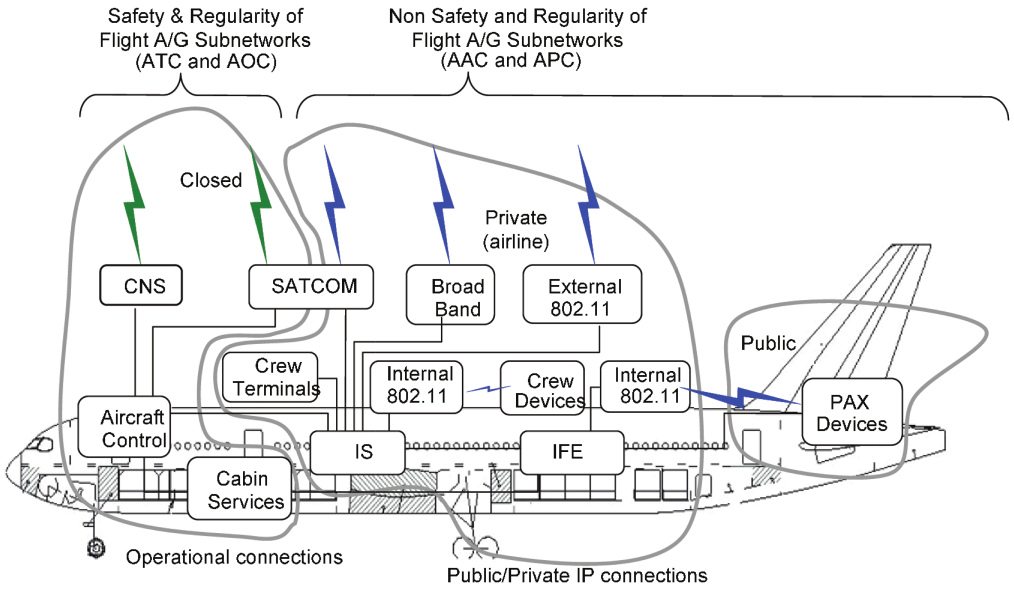

Figure I.4 – Types de communication du domaine avionique (ARINC 811)

En se qui concerne le réseau ATC/AOC, une étude menée par Eurocontrol et la FAA [Eurocontrol and FAA 2007] fournit une liste des différents services aéronautiques avec les performances attendues pour la transmission des informations. Ainsi, les communications ATC sont identifiées comme les plus critiques et obéissent à des réglementations bien spécifiques. Les communications APC, quant à elles, ne sont pas réglementées, car elles n'ont pas d'impact sur la sécurité-innocuité de l'appareil.

Un mécanisme de priorité et de préemption est nécessaire pour permettre aux communications tels que APC de coexister avec des communications primordiales pour la sécurité-innocuité de l'avion tels que l'ATC. Les communications qui ont une priorité inférieure peuvent être retardées ou interrompues si le canal est requis pour les communications de sécurité de priorité plus élevée.

Comme nous l'avons précédemment mentionné, cette approche ségrégative des communications dans le monde avionique permet d'assurer une immunité des systèmes avioniques vis-à-vis des malveillances. Cependant, cette approche peut s'avérer trop restrictive puisqu'elle ne permet pas une interaction entre entités appartenant à des domaines avioniques distincts.

Dans le cadre de ce manuscrit, nous nous intéressons aux interactions entre applications avioniques ayant des niveaux de criticités hétérogènes. Ainsi, ces applications appartiennent à des domaines avioniques disjoints, et possèdent des niveaux de DAL distincts également. Nous proposons de présenter dans la section suivante les différents types d'architectures avioniques utilisées au court du temps.

I.3.2 Architectures avioniques

Le terme architecture avionique représente l'ensemble des matériels et logiciels embarqués à bord de l'avion, qui assurent diverses fonctions telles que le traitement des informations provenant des capteurs, le pilotage automatique, la gestion du niveau de carburant, les échanges de messages avec l'opérateur au sol, pendant le vol, etc. Les architectures avioniques occupent une place non négligeable dans le coût des avions modernes (de l'ordre de 33% du coût total [PIPAME 2009]).

Historiquement, dans les systèmes avioniques, chaque fonction de contrôle disposait de ses propres ressources matérielles (représentées par une machine ou un calculateur) pour son exécution, comme le montre la Figure I.5. Ces dispositifs dédiés sont très souvent répliqués pour la tolérance aux fautes et peuvent varier d'une fonction à l'autre. Il en résulte ainsi une architecture hétérogène et faiblement couplée, où chaque fonction peut opérer de manière quasi indépendante vis-a-vis des autres fonctions. Une telle architecture est qualifiée de fédérée. Par exemple, la construction des Airbus A330 et A340 repose sur ce type d'architecture. Les équipements numériques mettant en œuvre certaines des fonctions à bord sont reliés par des bus mono-émetteur.

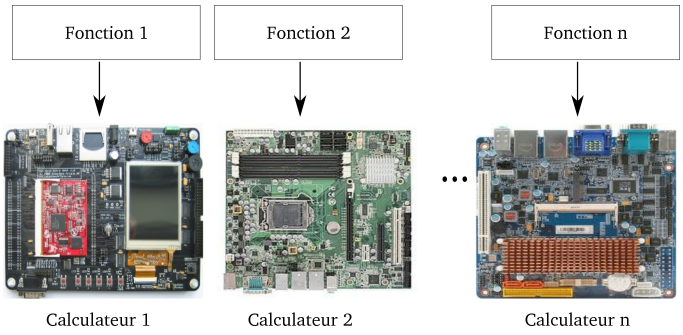

Figure I.5 – Architecture fédérée

Un avantage majeur des architectures féderées, en plus de leur simplicité, est la minimisation des risques de propagation d'erreurs qui peuvent survenir lors de l'exécution d'une fonction au sein du système. Cela assure une meilleure disponibilité du système global dans la mesure où une fonction défaillante peut éventuellement être isolée sans avoir recours à un arrêt complet du système. Cette caractéristique inhérente aux architectures fédérées est primordiale dans des systèmes critiques comme ceux qui sont embarqués dans les avions. Un autre avantage des architectures fédérées est leur hétérogénéité. En effet, les types de machines utilisées peuvent varier d'une fonction à l'autre au sein du même système. Cela permet l'utilisation de machines ayant des puissances variables.

Cependant, cette vision classique de systèmes fédérés présente des inconvénients liés à cette forte dépendance entre le logiciel et le matériel que nous résumons dans les points suivants :

- La communication entre systèmes dépend des applications et également du matériel d'exécution. Toute mise à jour d'une entité logicielle implique une modification importante dans le mécanisme de communication, entraînant souvent la modification des autres entités logicielles pour maintenir une bonne interopérabilité.
- Le matériel utilisé dans les applications avioniques est très hétérogène. En effet, chaque application possède ses propres contraintes fonctionnelles, et est exécutée

sur un matériel qui lui est dédié. Assurer la maintenance matérielle dans ces conditions est une tâche fort coûteuse, d'un point de vue économique.

- La durée de vie d'un avion est de l'ordre de plusieurs dizaines d'années (25-30 ans, voire plus). Cette durée de vie est longue relativement à la durée de vie d'un matériel. En effet, vu les avancées technologiques actuelles, un matériel devient obsolète au bout de quelques années seulement. Dans de nombreux cas, le matériel planifié durant la conception peut même devenir obsolète au moment de la construction de l'appareil. De plus, même si ce matériel est mis à jour, il faut bien s'assurer de la portabilité du logiciel qui n'a pas été originellement développé pour ce nouveau matériel. Cette tâche s'avère particulièrement coûteuse en raison du lien étroit qui existe entre le logiciel et le matériel dans un système fédéré.

Dans les années 1990, sous l'impulsion de la commission européenne, des projets européen de recherche tel que PAMELA, NEVADA et VICTORIA [SFP 2001b, SFP 2001a, SFP 2001c] , ont permis l'émergence d'une autre vision dans la conception des systèmes avioniques ; celle-ci a pour but de pallier les insuffisances des architectures fédérées en proposant de dissocier les composants logiciels des composants matériels (dans les systèmes fédérés). Ce type d'architecture est qualifié de modulaire intégré, plus communément appelé IMA (*Integrated Modular Avionics*) [ARINC 653 1997]. Le principe de l'architecture IMA [Morgan 1991] est de répartir dans tout le volume de l'appareil un ensemble de ressources (calcul, mémoire, communication), qui pourront être partagées par plusieurs applications, qui accompliront les différentes fonctions du système avionique. Des exemples d'avions adoptant la solution intégrée sont l'Airbus A380 ou le Boeing B777.

L'approche IMA présente trois principaux avantages :

1. Réduction du poids grâce à un plus petit nombre de composants matériels et un câblage réduit, ce qui augmente l'efficacité énergétique ;
2. Coût d'entretien réduit par l'utilisation de modules génériques ;
3. Réduction des coûts de développement par l'utilisation de systèmes standardisés.

Cette organisation du système présente une plateforme unique (*cf.* Figure I.6) pour toutes les applications logicielles avioniques qui permet d'économiser les ressources, contrairement aux architectures fédérées. Par la même occasion, le coût de réalisation peut être limité de façon raisonnable.

La plateforme consiste en un réseau temps-réel distribué, permettant à différentes applications de s'exécuter en parallèle. Cependant, elle introduit un risque de propagation d'erreur. Par exemple, une fonction en dysfonctionnement peut monopoliser le système de communication ou envoyer des commandes inappropriées, et pour chacune des fonctions il est difficile de se mettre à l'abri d'un tel comportement. Il est donc indispensable qu'une telle architecture assure un partage de ressource sûr entre les applications : le mécanisme dit de partitionnement [ARINC 653 1997]. Par ce moyen, une ou plusieurs applications regroupées au sein d'un même module peuvent s'exécuter de manière sûre (deux fonctions quelconques prévues pour s'exécuter dans un même module ne peuvent en aucun cas interagir l'une sur l'autre).

De manière pratique, l'architecture physique est décrite par la norme ARINC 651. Les ressources sont regroupées dans des modules génériques appelés LRM (Line Replaceable Module), qui sont à leur tour regroupés dans des étagères, la communication au sein de ces étagères étant réalisée avec des bus spéciaux, généralement de type ARINC 659. Les modules peuvent être de trois types :

- des modules cœurs qui sont ceux qui se chargent de l'exécution des applications ;
- des modules d'entrée/sortie permettant la communication avec des éléments ne respectant pas l'architecture IMA ;
- des modules passerelles servant à la communication entre étagères (l'architecture IMA n'impose pas de moyen de communication spécifique entre ses différents composants).

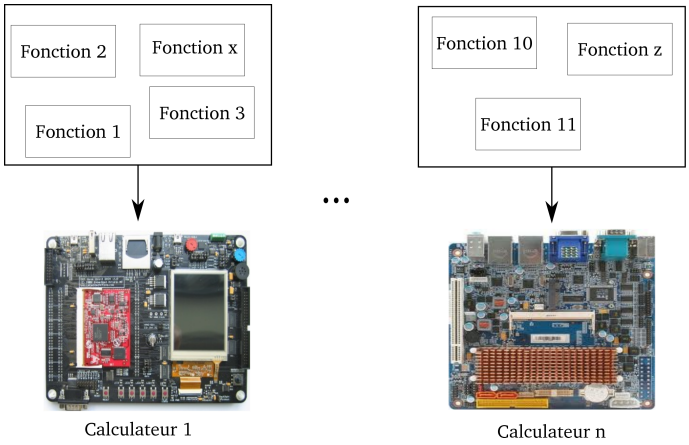

Figure I.6 – Architecture IMA

Ce partitionnement est classé en partitionnement spatial (par exemple, la gestion de la mémoire) et temporel (la gestion des ressources temporelles par la CPU) [John 1999]. Le partitionnement spatial des applications repose sur l'utilisation de composants matériels chargés de gérer la mémoire pour empêcher toute corruption de zones mémoires adjacentes à une zone en cours de modification. Le partitionnement temporel dépend des fonctionnalités attendues de l'ensemble (par exemple, on aura besoin d'exécuter plus souvent des fonctions qui s'occupent du rafraichissement de paramètres critiques). Le partitionnement désigne un mécanisme mis en place pour assurer une gestion des ressources qui ne met pas en péril le fonctionnement des applications qui l'utilise.

Une architecture IMA peut donc être perçue comme un gestionnaire de ressources matérielles pour des applications avioniques embarquées. Ces applications sont implantées dans des modules pouvant avoir des niveaux de criticité différents. L'architecture IMA se doit d'être sûre afin de permettre un tel partitionnement entre applications à criticités multiples. Dans [ARINC 653 1997] le partitionnement est défini comme devant assurer une isolation totale.

L'isolation est réalisée en introduisant des partitions qui sont des espaces d'exécution logiques attribués à chaque fonction avionique. L'accès aux ressources (CPU, mémoire, etc) est ensuite contrôlé pour qu'il n'y ait aucune altération du fonctionnement des entités les plus critiques (des entités du niveau DAL-A ou DAL-B, par exemple). Une telle architecture est similaire à l'architecture d'un système d'exploitation classique.

En effet, un système d'exploitation offre aux différentes applications qui y sont installées de s'exécuter en parallèle, indépendamment de la couche matérielle. Une architecture IMA (Figure I.6), tout comme un système d'exploitation, offre des services (par exemple service de communication) aux applications qui y sont installées. Ces services sont accessibles via une interface de programmation des applications (Application Programming Interface : API) unique pour toutes les fonctions avioniques. Ainsi, une plateforme

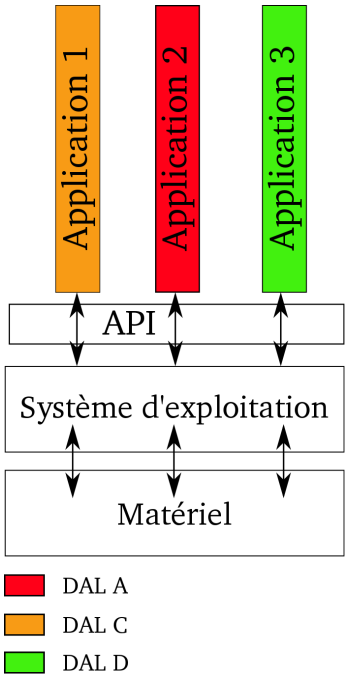

Figure I.7 – Exemple de Module IMA

IMA est constituée d'un certain nombre de modules distribués dans l'avion. Ces modules sont connectés entre eux et également avec des périphériques de l'avion (typiquement des périphériques d'entrée/sortie comme des capteurs ou des interfaces homme machine).

La Figure I.7 présente l'aspect en couche d'un module IMA. Le module est constitué d'un système d'exploitation en charge d'exécuter plusieurs partitions. La communication entre les partitions et le système d'exploitation du module se fait via l'API. Cette interface de programmation est la seule couche visible pour une application dans une partition donnée. Cette abstraction présente les avantages suivants :

- Du fait de la transparence du matériel vis-à-vis des fonctions avioniques, il est possible d'utiliser du matériel distinct afin d'assurer une diversification matérielle (*cf.* I.2), sans conséquence pour le fonctionnement de l'application.
- La robustesse du partitionnement fait qu'une application peut être développée et testée d'une façon incrémentale, sans que cela n'altère le fonctionnement des autres partitions.
- Grâce à l'interopérabilité des modules, en cas de défaillance d'un module, il est possible de reconfigurer une application logicielle pour qu'elle s'exécute sur un second module, ce qui présente une propriété fort intéressante en terme de disponibilité de l'application.

Les architectures IMA sont déjà utilisées pour des fonctionnalités spécifiques sur certains appareils. Par exemple, le système d'exploitation PikeOS [Kaiser & Wagner 2007] est utilisé pour son niveau de certification afin de construire une architecture facile à certifier. Les équipements ainsi produits pourront être déployés sur le prochain A350 XWB . Cependant, la certification des systèmes avioniques utilise toujours la vision des systèmes fédérés. Des projets de recherche tel que SCARLETT [SFP 2011] définissent les évolutions de l'architecture IMA afin d'améliorer entre autre, le processus de certification.

Comme nous avons pu le constater, la sécurité, au sens innocuité, est l'une des premières exigences qui a guidé le développement des fonctions avioniques (aussi bien au niveau matériel que logiciel). Cependant, la composante sécurité-immunité est devenue de plus en plus présente dans le monde avionique.

I.3.3 Sécurité-immunité pour l'avionique

Nous proposons de détailler dans cette section quelques pistes de réflexion sur l'aspect immunité. Ces pistes sont encore en cours d'analyse et d'investigation dans le monde avionique.

Depuis de nombreuses années, le développement des architectures avioniques intègre des composants issus des technologies de l'information grand public (typiquement des composants sur étagère ou aussi communément identifiés par le vocable COTS : *Commercial-Off-The-Shelf*). Ce phénomène est également identifié dans l'étude prospective [HLGAR 2011]. L'utilisation croissante des composants COTS est accompagnée d'un besoin de communication bidirectionnelle entre les modules avioniques et le monde ouvert (autrement dit tous systèmes n'appartenant pas au monde avionique). Notons que le terme avionique doit être pris ici dans son sens le plus large, couvrant les systèmes embarqués à bord de l'avion dont les fonctionnalités assurent le contrôle ou la commande de l'appareil. De plus, comme nous l'avons mentionné au paragraphe I.3.1, les différents domaines avioniques prennent en compte ce genre de communication [ARINC 811 2005].

L'utilisation des fonctionnalités étendues qu'offrent les applications COTS par des modules avioniques peut nécessiter une forte interaction avec le monde ouvert. Cependant, les applications COTS n'ont pas été développées en respectant les contraintes du type DO-178 vue à la Section I.3.1. Il convient donc de définir de nouvelles règles afin de permettre l'utilisation d'applications COTS dans l'industrie aéronautique.

Nous présentons dans un premier temps les notions de criticité et de confiance afin de faciliter la lecture des paragraphes suivants. Ensuite nous décrirons les différents travaux et leurs évaluations qui ont mené à la définition des Critères Communs. Nous détaillons ensuite un modèle de communication pouvant être utilisé dans l'industrie aéronautique.

Les Notions de criticité et de confiance

Comme nous l'avons indiqué dans la Section I.3.1, les systèmes avioniques sont critiques. Ils sont conçus afin de réaliser une mission spécifique. Nous désignons par « tâche » la fonction du système en charge d'accomplir la mission requise. Pour être concrétisée, une tâche doit être implémentée sur un support d'exécution que nous désignons par « module ». Un module peut être de nature logicielle ou matérielle (ou les deux). Nous proposons de spécifier dans ce qui suit les notions de criticité et de confiance associées à une tâche.

La criticité est associée à une tâche en charge de réaliser une mission donnée. Une tâche est considérée critique si, en cas de défaillance de cette tâche, des conséquences catastrophiques peuvent arriver au système. Cette définition correspond à la terminologie introduite dans les normes avioniques. Par « conséquences catastrophiques » nous désignons

les blessures physiques des utilisateurs humains du système. Ainsi, le crash d'un avion est considéré comme un événement catastrophique. La gravité de la défaillance d'une tâche peut dépendre des différentes phases de la mission attribuée à la tâche. Ainsi, la tâche de commande d'un avion est très critique lorsque l'avion est en phase de vol. Cette même tâche devient moins critique dès lors que l'avion est au sol. Cependant, même si une tâche à une criticité variable en fonction des phases de la mission, c'est la criticité la plus élevée qui doit être prise en compte dans l'étude du système global. Dans l'exemple que nous avons donné, la tâche sera donc considérée comme critique lors de la conception et du déploiement du système final.

Ainsi, afin de définir le niveau de criticité d'une tâche, différentes études doivent être réalisées pour évaluer le risque subi par les utilisateurs en cas de défaillance de cette tâche. Ces études fournissent un ensemble de scénarios identifiant les défaillances potentielles de la tâche et, en fonction de la gravité des défaillances, un niveau de criticité est attribué à la tâche.

L'évaluation de la criticité est un processus réalisé en prenant en compte les aspects environnementaux d'exécution de la tâche. En effet, la gravité d'une défaillance dépend essentiellement de la nature de la tâche, mais également des différentes interactions entre tâches. Il est donc primordial de spécifier les interactions d'une tâche avec son environnement (qui comprend tous les éléments extérieurs à la tâche) afin de lui attribuer le niveau de criticité correspondant.

Le niveau de criticité est une propriété propre à la tâche. La diminution du niveau de criticité d'une tâche implique que l'on considère que la gravité de sa défaillance a diminué. Cette diminution doit être justifiée.

Nous rappelons qu'une tâche est implémentée sur un module (logiciel ou matériel). Ainsi, il est donc indispensable pour une tâche critique, de s'exécuter sur un module de confiance. Le niveau de confiance d'un module traduit la fidélité des spécifications requises par la tâche. Par conséquent, le niveau de confiance qui caractérise un module doit être en adéquation avec le niveau de criticité de la tâche qu'il réalise [2].

Prenons l'exemple d'une tâche en charge de calculer une vitesse moyenne, en fonction de valeurs mesurées. Le module qui effectue la tâche est dit de confiance s'il réalise correctement le calcul, mais également si ses capteurs mesurent correctement les valeurs. Dans cet exemple, la tâche peut être divisée en deux sous-tâches : une pour la capture de la valeur de la vitesse et une pour la réalisation du calcul. Il est donc possible d'utiliser deux modules afin de réaliser chaque sous tâche. Dans ce cas, le module de calcul est de confiance si à partir de mesures correctes, il fournit un calcul correct. Cependant, pour la réalisation de la tâche (mesure + calcul), il est nécessaire que le module de mesure soit de confiance également. À défaut, le résultat final peut être incorrect, ce qui ne répond pas aux besoins de criticité de la tâche. Ainsi, il faut s'assurer que tous les modules associés à une tâche soient tous d'un niveau de confiance en adéquation avec le niveau de criticité de cette tâche.

Le niveau de confiance d'un module dépend de trois principaux paramètres :

- La crédibilité, qui traduit la confiance que l'on accorde à la source de données

2. Afin de faciliter la compréhension de la Figure I.8, nous considérons le niveau de confiance égal au niveau de criticité.

utilisées en entrée du module durant la phase opérationnelle ;

- La validation, qui correspond aux efforts fournis durant la phase de conception et de développement du module afin de satisfaire les exigences de la tâche implémentée ;
- L'intégrité, qui consiste à assurer l'absence d'altération du module durant la phase opérationnelle.

La Figure I.8 présente les différents niveaux introduits dans cette section qui se résument comme suit : à une tâche est affecté un niveau de criticité en fonction de la gravité de la défaillance de cette tâche. Pour satisfaire ce niveau de criticité, la tâche est exécutée sur un module auquel est attribué un niveau de confiance. Cette confiance dépend de la phase de construction (niveau de validation), des données reçues à l'exécution (niveau de crédibilité) et de la non-altération de ce module durant la phase opérationnelle (niveau d'intégrité).

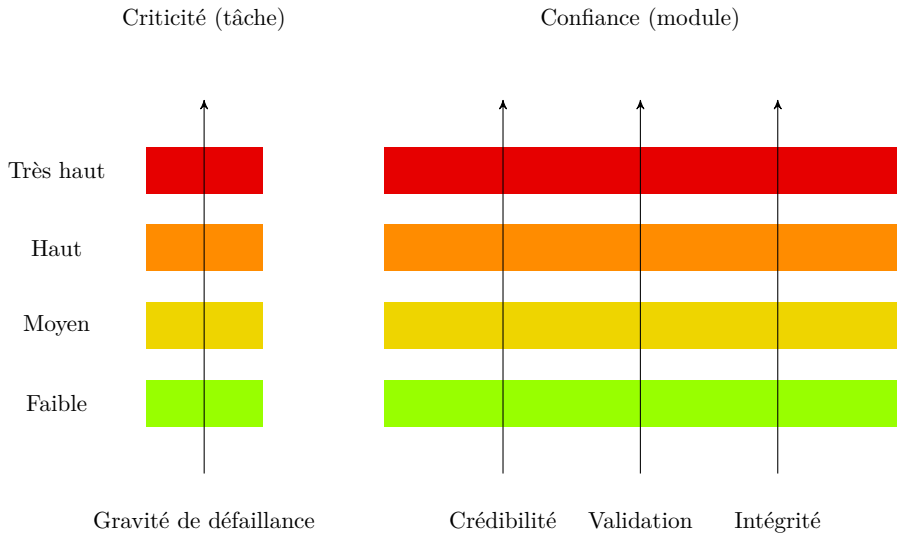

Figure I.8 – Vision globale des niveaux de criticité et de confiance

La combinaison des notions de criticité et de confiance aboutit à la notion de sécurité. Cette dernière a fait l'objet de plusieurs. études Elles sont présentées dans la section suivante.

Évolution vers les Critères Communs

Les premiers travaux ont été initié par le conseil scientifique de l'armée américaine (*Defense Science Board*) afin d'étudier les différentes approches de protection des données classifiées dans les systèmes informatiques. Parmi leurs productions, le document *Trusted Computer Security Evaluation Criteria* (TCSEC) datant de 1985, plus connu sous le nom de « livre orange », est devenu une norme du Département de la Défense américaine [TCSEC 1985]. Ce document est la référence en matière d'évaluation de la sécurité des systèmes informatiques. Les critères offrent une classification à sept niveaux de sécurité (A1, B1, B2, B3, C1, C2, D), regroupés en quatre classes (A, B, C, D). Quatre familles de critères sont définies pour chaque niveau. Elles traitent respectivement de la politique d'autorisation, de l'audit, de l'assurance et de la documentation. L'évaluation d'un produit consiste à lui attribuer un des sept niveaux de sécurité. Cette attribution n'aboutit que si le produit répond à tous les critères du niveau en question. La politique d'autorisation employée ne permet pas de prendre en compte les contraintes liées aux pratiques commerciales. Cette limite a suscité la création de nouveaux critères dans d'autres pays, avec, par exemple, les *Canadian Trusted Computer Product Evaluation Criteria* [CTCPEC 1993] (CTCPEC) et les *Japanese Computer Security Evaluation Criteria* [JCSEC 1992]. Nous citons ci-dessous l'exemple des *Information Technology Security Evaluation Criteria* (ITSEC) adoptés par la Communauté Européenne et des « Critères Communs »(CC).

Les ITSEC sont le résultat de l'harmonisation de travaux réalisés au sein de quatre pays européens : l'Allemagne, la France, les Pays-Bas et le Royaume-Uni [ITSEC 1991]. Les ITSEC se différencient essentiellement du livre orange par la définition d'un certain nombre de classes de fonctionnalités d'une part et un certain nombre de classes d'assurance d'autre part. Une classe de fonctionnalité décrit les mécanismes que doit mettre en œuvre un système pour être évalué à ce niveau de fonctionnalité. Une classe d'assurance permet, quant à elle, de décrire l'ensemble des preuves qu'un système doit apporter pour montrer qu'il implémente réellement les fonctionnalités qu'il prétend assurer. Les ITSEC utilisent le terme cible d'évaluation (*Target of Evaluation* ou TOE). Le contenu d'une cible d'évaluation comprend : une politique de sécurité, une spécification des fonctions requises dédiées à la sécurité, une définition des mécanismes de sécurité requis et le niveau d'évaluation visé.

Les Critères Communs (CC) [CC 3.1 2009a] sont nés de la tentative d'harmonisation des critères canadiens (CTCPEC), des critères européens (ITSEC) et des critères américains (TCSEC). Ils contiennent deux parties bien séparées comme dans les ITSEC : fonctionnalité et assurance. De méme que dans les ITSEC, les CC définissent une cible d'évaluation (TOE) et une cible de sécurité (*Security Target*). Une cible d'évaluation désigne le système ou le produit à évaluer. La cible de sécurité contient les objectifs de sécurité d'une cible d'évaluation particulière. La cible de sécurité pour une cible d'évaluation représente la base d'entente entre développeurs et évaluateurs. Elle peut contenir les exigences d'un ou de plusieurs profils de protection (*Protection Profiles*) prédéfinis. Une des différences essentielles entre ITSEC et CC réside dans l'existence de ces profils, qui avaient auparavant été introduits dans les Critères Fédéraux [FCITS 1991]. Un profil de protection définit un ensemble d'exigences de sécurité et d'objectifs, indépendants d'une quelconque implémentation, pour une catégorie de cible d'évaluation.

La Figure I.9 issue de [CC 3.1 2009a] décrit les différentes coopérations qui ont permis la réalisation de cette norme.

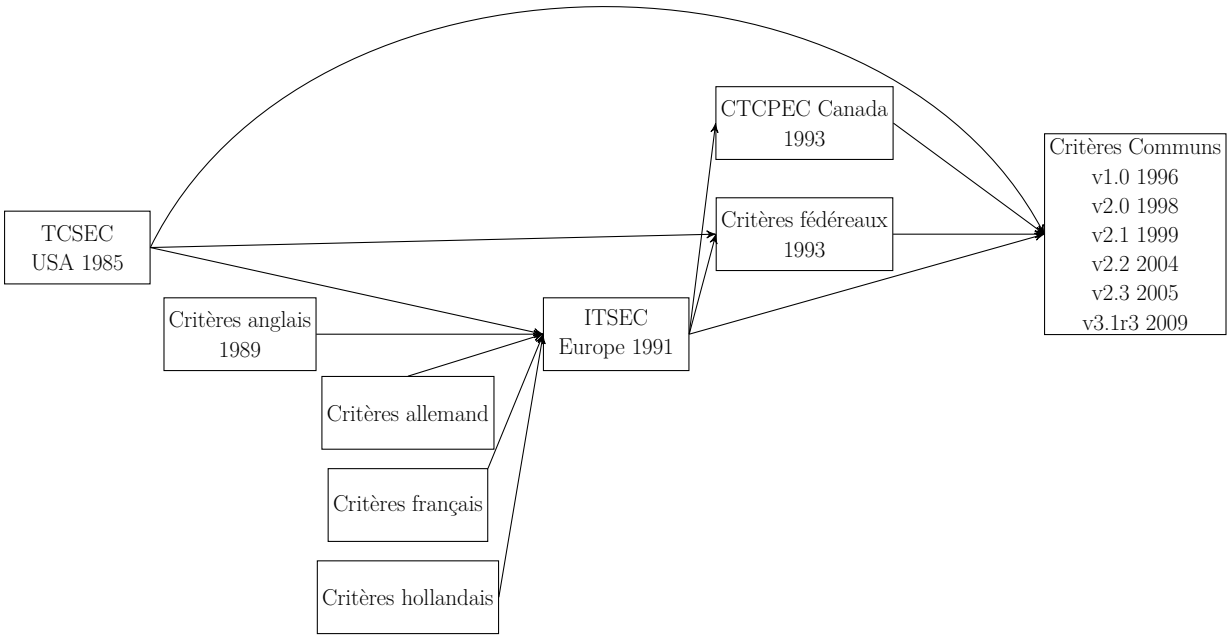

Figure I.9 – Historique des Critères Communs

La méthodologie définie par la version actuelle des **CC** (3.1) [CC 3.1 2009a], est conçue pour répondre à des fonctionnalités de sécurité ainsi que l'assurance de ces fonctionnalités. Les exigences d'assurance sont organisées en six classes :

- ALC (Life Cycle support Activity) : Activité en relation avec le support de développement, de la configuration au produit livré.
- AGD (Guidance Documents Activity) : Activité qui permet la préparation des documents (guides, manuels,...) pour l'installation et l'utilisation du produit.
- ADV (Development Activities) : Activité définissant les étapes de développement du produit.
- ASE (Security target Evaluation Activity) : activités liées au contenu de la cible de sécurité (description de la cible de sécurité, objectifs de sécurité, fonctions de sécurité, activités d'assurance, ...)
- ATE (Tests Activities) : Activité relative aux tests du produit.
- AVA (Vulnerability Assessment Activity) : Evaluation des vulnérabilités

Chaque classe est ensuite décomposée en différentes familles. Par exemple, la classe AVA contient la famille AVA_VAN qui rassemble les composants permettant l'évaluation des vulnérabilités. Les niveaux d'assurance sont répartis en fonction de la finesse et de la rigueur utilisés pour décrire les éléments qui sont nécessaires pour effectuer l'évaluation. Une évaluation du niveau d'assurance ou EAL (*Evaluation Assurance Level*) définit l'ensemble des composants qui doivent être appliqués dans chaque famille. Les CC offrent ainsi différents niveaux d'assurance EAL : de EAL1, le niveau le plus bas de l'assurance, à EAL7, le niveau le plus haut.

Critères Communs et niveaux de sécurité

Dans [Blanquart *et al.* 2012], les auteurs ont étudié les relations qu'il pouvait y avoir entre les niveaux de sécurité-immunité tels que définis au paragraphe I.3.1(c'est à dire les DAL) et les EAL. Un niveau de sécurité est alloué à une fonction après une analyse des risques. Une analyse de risque de sécurité-immunité définie par la norme [ISO 27005 2008] se compose de cinq étapes principales, comme le montre la Figure I.10 :

1. Définition du contexte : permet de délimiter le périmètre de l'analyse du système et définir les différents critères d'évaluation et d'acceptation.
2. Identification des risques : liste les actifs qui doivent être pris en compte dans l'évaluation des risques de sécurité. Le terme « actif » dans le domaine de la sécurité des équipements de vol, signifie un équipement ou partie de celui-ci qui peut être attaqué et de ce fait susceptible de corrompre les équipements de vol. Ensuite, les évènements qui ont un impact sur la sécurité des actifs sont définis afin d'élaborer des scénarios de menace.

   Les risques qui découlent des scénarios de menace sont inacceptables lorsque leur probabilité d'apparition n'est pas tolérable par le client, par exemple si le nombre de fois que la menace est susceptible de se produire est trop élevé par rapport au critère d'acceptation défini. Si les risques sont inacceptables, des mesures de réduction des risques doivent être définies pour modifier ou compléter la conception. L'acceptation des risques devra ensuite être examinée après la mise à jour de l'évaluation des risques à envisager par la modification de la conception. Ceci peut être répété jusqu'à ce qu'un risque acceptable soit trouvé.
3. Appréciation des risques : cette étape se compose de deux phases, l'estimation et l'évaluation des risques. Ces phases définissent l'impact des risques issus des scénarios de menace sur les actifs et permettent d'identifier les vulnérabilités utilisées pour exécuter le scénario ainsi que les contre-mesures de sécurité existantes.
4. Traitement des risques : le traitement du risque peut modifier la conception, ou ajouter des contre-mesures de sécurité qui ont une efficacité suffisante pour réduire le risque à un niveau acceptable. Les niveaux de sécurité sont utilisés pour classer l'efficacité nécessaire pour atteindre un niveau acceptable.
5. Acceptation des risques : cette étape clôture l'analyse.

Le niveau de sécurité associé à une fonction tel que défini par [ED 202 2010] peut être considéré comme représentatif de sa capacité à réduire la probabilité de réussite des attaques. Cette probabilité se compose de l'**efficacité**, la capacité du mécanisme de sécurité à protéger le système contre une attaque, et de la **conformité**, la confiance apportée au mécanisme de sécurité.

Un niveau de sécurité croissant allant de E à A peut être affecté à des fonctions qui se produisent dans un scénario de menace. La règle d'attribution du niveau de sécurité est basée sur la réduction de la probabilité ou la fréquence vraisemblable du scénario de menace à un niveau acceptable. Le document [ED 202 2010] propose cinq valeurs : fréquente, probable, distante, très distante et très improbable.

Par exemple, considérons une architecture contenant une fonction pour laquelle la probabilité d'une menace est fréquente. Si cette fonction est capable de réduire la prob-

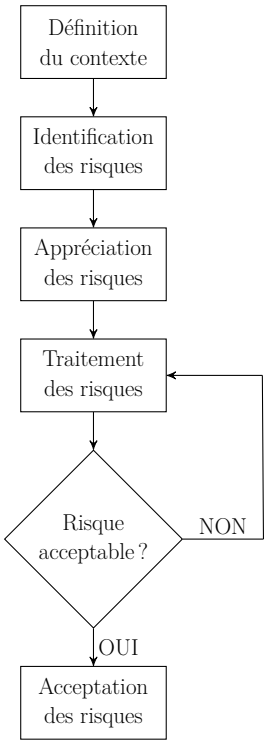

Figure I.10 – Diagramme d'analyse de risque

abilité de la menace de « probable » à « très improbable », soit une réduction de trois niveaux, alors il pourra lui être associé le niveau de sécurité B.

Le Tableau I.1 présente une attribution des niveaux de sécurité en fonction de la réduction de probabilité.

Réduction de probabilité	0	1	2	3	4
Niveau de sécurité	E	D	C	B	A

Tableau I.1 – Règle d'attribution de niveau de sécurité

Comme nous venons de le présenter, le niveau de sécurité est utilisé pour sélectionner les exigences d'assurance adéquates en termes d'efficacité et de conformité. Nous avons vu au paragraphe I.3.3 que les CC disposent de niveaux d'assurance. Les auteurs de [Blanquart *et al.* 2012] ont proposé d'établir une correspondance entre niveaux d'assurance et niveaux de sécurité. Plus précisément, ils rapprochent les niveaux de sécurité avec les AVA_VAN, composants permettant l'évaluation des vulnérabilités des **CC**. Les auteurs de l'étude s'appuient sur la correspondance faite dans [CC 3.1 2009b] entre les évaluations de vulnérabilités (AVA_VAN) et les niveaux d'assurance (EAL).

Tout d'abord, en se basant sur le fait que les évaluations de niveau EAL5 ne sont pas très fréquentes, ils font l'hypothèse de ne pas inclure dans leur analyse les niveaux EAL6 et EAL7. La majorité des cibles de leur étude sont des équipements embarqués sur une plateforme complexe, ils suggèrent donc de limiter le niveau AVA_VAN à AVA_VAN.4.

Le Tableau I.2 issue de [Blanquart *et al.* 2012] montre une possible correspondance entre niveau de sécurité et EAL. Un inconvénient majeur qui a été identifié est la non distinction entre les niveaux de sécurité A et B.

Niveau de sécurité	E	D	C	B	A
AVA	1	2	3	4	4
EAL	1	2,3	4	5	5

Tableau I.2 – Possible correspondance entre niveaux de sécurité et EAL

Nous noterons également qu'une telle correspondance est sujette à discussion par l'utilisation de nombreuses hypothèses. Toutefois, elle a pour mérite d'initier la mise en place de passerelles pragmatiques permettant d'assurer la correspondance entre les normes de sécurité.

I.3.4 Modèle Totel

La section précédente à présenté la sécurité-innocuité pour les système embarqués. En particulier, les systèmes ont été caractérisés par leur niveau de criticité. Or dans un système plus complexe, comportant différent niveaux de criticité, il peut être nécessaire d'avoir une communication bidirectionnelle entre ces niveaux. Dans la suite de cette section nous présentons un modèle pour assurer de telles communications.

Le modèle proposé dans [Totel 1998] supporte l'exécution des tâches de différents niveaux de confiance sur un seul système. Le modèle Totel considère des flux d'information entre des objets dont le niveau de confiance est différent. La Figure I.11 présente l'architecture globale qui doit être déployée pour permettre la communication bidirectionnelle entre des objets ayant des niveaux de confiance différents. Nous rappelons que la relation entre confiance et criticité est décrite sur la Figure I.8.

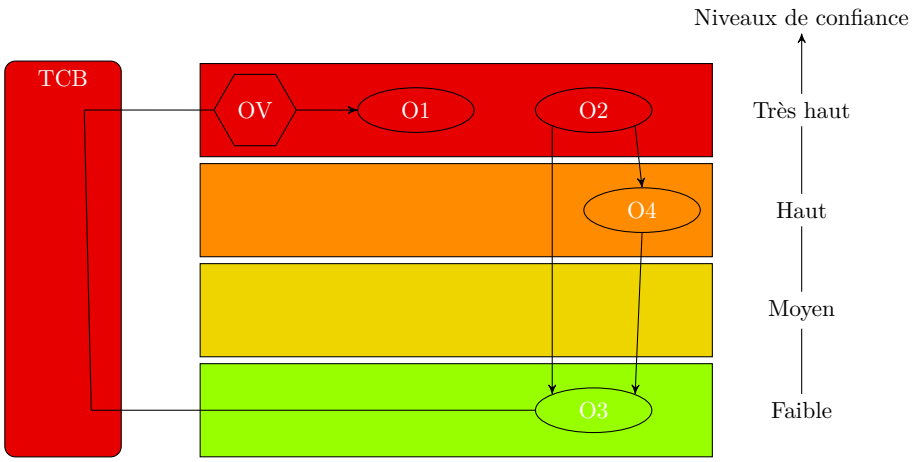

Figure I.11 – Modèle Totel

Les objets peuvent être des composants logiciels ou des sous-systèmes de matériel/-logiciel. La direction des flèches correspond à la direction du flux logique de l'information.

La communication descendante n'affecte pas défavorablement les tâches de haut niveau de confiance, ainsi elle n'a pas besoin de vérification spécifique et est implicitement autorisée. Au contraire, la communication ascendante directe est interdite. La seule manière pour un objet de haut niveau de confiance de recevoir des informations d'un objet de plus faible niveau de confiance est d'utiliser un canal de communication spécifique. Ce canal doit être contrôlé par une « base informatique de confiance » en anglais : *Trusted Computing Base* (TCB), qui offre un point d'entrée spécifique et un point de sortie correspondant pour chaque flux de communication ascendant. Pour éviter une quelconque violation de ce principe, la TCB contrôle toutes les communications ascendantes dans le système et empêche les flux ascendants non autorisés. Seul les flux ascendants autorisés par la TCB sont permis et chacun de ces flux doit être adressé à un Objet de Validation (OV).

Un OV est un mécanisme permettant de vérifier les messages avant de les adresser aux objets de destination. La mise en œuvre d'un OV est extrêmement liée au type d'information qu'il vérifie. En pratique, un OV utilise un mécanisme de tolérance aux fautes (par exemple, un algorithme de décision) pour augmenter le niveau de confiance des données issues d'objets de plus faible confiance (par exemple, O3). Cette garantie dépend des hypothèses faites sur les fautes qu'un OV doit tolérer.

La TCB est constituée d'un canal de communication ascendant, d'un noyau de gestion des ressources et d'un noyau d'intégrité. Le noyau de gestion des ressources est utilisé pour contrôler les ressources (par exemple, la mémoire, le processeur, etc.) partagées par les différentes tâches et empêcher des tâches de consommer les ressources nécessaires à des tâches plus critiques. Le noyau d'intégrité met en œuvre des mécanismes de vérification d'intégrité qui assurent que tous les flux de l'information sont sûrs. La TCB doit interagir avec les objets les plus critiques du système, donc elle doit être validée au niveau de confiance le plus haut.

Dans la Figure I.11, l'objet O1 qui, avec O2, est un des deux objets les plus critiques du système, requiert des informations provenant d'un objet de niveau inférieur, l'objet O3. Nous devons nous assurer que les informations venant des niveaux inférieurs n'altèrent pas le comportement de l'objet O1. A cette fin, les informations venant de l'objet O3 doivent être validées par des Objets de Validation (OV) associés à l'objet O1, dans le but d'augmenter le niveau de confiance placé sur les données provenant de l'objet O3 en direction de l'objet O1. Les objets de validation sont des éléments cruciaux dans ce modèle et doivent ainsi être du même niveau de confiance que l'objet de destination. Ils sont les seuls objets dans le modèle autorisé à recevoir des données de niveaux de confiance inférieurs.

Conclusion

Ce premier chapitre a présenté le contexte de nos travaux concernant les architectures embarquées sécurisées pour le domaine avionique. Plus particulièrement, notre objectif est d'assurer la sécurité des systèmes critiques, omniprésents dans l'industrie aéronautique.

Pour ce faire, nous avons tout d'abord présenté la terminologie propre au domaine de la sûreté de fonctionnement, avant de détailler dans la Section I.2 les mécanisme utilisés pour la tolérance aux fautes. Nous avons introduit par la suite les différentes normes et standards utilisés pour le développement des fonctions avioniques. Nous avons présenté à titre d'exemple différentes architectures. Tout d'abord, les architectures fédérées qui permettent d'implémenter une fonction sur un module matériel. Les architectures IMA offrent des mécanismes de séparation spatiales et temporelles. Il est ainsi possible pour un même module matériel d'exécuter différentes fonctions de niveaux de criticité différents. Ces architectures nous ont permis de discuter des aspects sécurité-immunité dans les appareils actuels et futurs et de présenter les études en cours sur la complémentarité des normes et standards. L'évolution des systèmes avioniques nécessite l'autorisation d'une communication bidirectionnelle entre fonctions de niveaux de criticité différents. Afin d'assurer de telles interactions, un modèle de communication nommé modèle Totel a été décrit.

La présentation de l'évolution d'une architecture d'équipements mobiles dans le chapitre suivant nous permet de détailler le cas d'étude utilisé lors de nos travaux et ainsi de décrire les études réalisées.

Chapitre II

Équipements mobiles pour systèmes avioniques

Introduction

Les systèmes informatiques qui réalisent différentes fonctions à bord de l'avion sont de plus en plus complexes et nombreux, notamment pour améliorer les conditions de travail de l'équipage et également satisfaire les exigences des passagers. Nous emploieront le terme « systèmes avioniques » pour désigner de tels systèmes dans la suite de ce manuscrit.

Dans ce chapitre, nous présentons tout d'abord deux équipements électronique que sont l'*Electronic Flight Bag* (EFB) et le portable de maintenance (PMAT *Portable Maintenance Access Terminal*). Par la suite nous détaillons le scénario actuel de maintenance ainsi qu'une proposition d'évolution. Puis, nous décrivons l'architecture logicielle mise en place pour la réalisation de ce scénario. Enfin, nous discutons des différentes améliorations à apporter à cette architecture.

II.1 Équipements mobiles

Les systèmes avioniques, tels que le système de navigation ou le système de commande de vol, s'appuient sur un grand nombre de données. Ces données sont issues des différentes sources (par exemple l'aéroport, la compagnie aérienne) et sont généralement transmises sous format papier, ce qui représente environ 18kg de documentation [Skaves 2011].

Depuis 1998 [ADR Florida 2012], des sociétés proposent des équipements permettant d'obtenir une partie de la documentation de bord sous format électronique. Le développement de ces équipements nommé EFB pour *Electronic Flight Bag* est encadré par des guides tel que [FAA 2003a] et [FAA 2003b] afin d'obtenir les autorisations d'utilisations par les autorités de certifications [1]. Un EFB est un dispositif contrôlé par la compagnie et

1. Ces autorités sont spécifiques aux pays du constructeur. Aux États-Unis, l'autorité compétente est

généralement assigné à une personne ou un avion. La compagnie doit s'assurer qu'aucun changement non autorisé aussi bien sur le matériel que sur le logiciel ne soit effectué et que sa maintenance et sa configuration soient correctement réalisées.

La première application attribuée à un EFB est la réduction du poids de la documentation papier que l'équipage de bord doit avoir dans l'avion (par exemple le manuel relatif à l'avion, la checklist ou la carte d'approche). Les compagnies aériennes souhaitent augmenter les capacités de ces équipements afin d'accroitre leurs utilisation. Par exemple, l'EFB pourrait être utilisé pour afficher des informations du trafic aérien, communiquer des données avec les stations au sol ou être connecté à Internet. Ces améliorations posent le problème des risques liés à la perte ou au dysfonctionnement de l'EFB.

Comme décrit précédemment, un EFB est principalement destiné à être utilisé dans la cabine de pilotage par le pilote. Un EFB peut être mobile ou fixe et doit être accessible par le pilote de manière à ne pas perturber son travail. Les EFB sont définis en trois classes principales.

Les EFB de classe 1 sont des ordinateurs portables de type COTS utilisés pour les opérations d'exploitation. Ils sont indépendants des systèmes de bord, ils ne se connectent pas au système d'information de l'avion et par conséquent ils n'ont pas besoin d'approbation opérationnelle. Les EFB de classe 2 reprennent les spécifications des EFB de classe 1 en ajoutant une connectivité aux systèmes de bord pendant les opérations d'exploitations. Cette connectivité requiert une approbation opérationnelle. Les EFB de classe 3 sont quant à eux des équipements non portables installés à bord de l'avion qui nécessitent une certification.

Comme nous venons de le décrire les EFB sont utilisés uniquement par le pilote afin de l'aider dans la réalisation de ses tâches. Par conséquent, les EFB ne peuvent pas être utilisés dans le cadre d'opérations de maintenance. Pour ce faire, un dispositif dédié est utilisé, le PMAT (*Portable Maintenance Access Terminal*). L'équipement que nous décrivons par la suite n'est toutefois pas encore utilisé dans les opérations de maintenance.

Le PMAT se définit comme un équipement portable muni d'un écran et d'un clavier utilisé pour les opérations de maintenance. Il a un accès distant à l'OMS, (On-board Maintenance System) que nous présentons brièvement dans la Section II.2.1, à travers différents ports à l'intérieur de l'avion. L'objectif de cet équipement est de permettre la réalisation des opérations de maintenance aussi bien à l'extérieur qu'à l'intérieur de l'avion. Par exemple, si une action de maintenance est requise sur les freins de l'avion, le PMAT doit se connecter au port dédié à l'OMS situé généralement au niveau des roues. A partir de là, le test peut être mené afin d'isoler la faute et permettre à l'opérateur d'effectuer les actions de maintenance. Enfin un test peut être réalisé pour vérifier l'opération.

Le PMAT peut être un ordinateur portable de type COTS, à condition d'implémenter une interface de communication compatible avec l'OMS. L'ordinateur de maintenance a une connexion au domaine AISD défini au Chapitre I (Figure I.2), ce domaine bien que moins critique que l'ACD a une criticité importante. Par conséquent, l'utilisation du PMAT comporte des risques au niveau de la sécurité-immunité de l'appareil. Bien que le PMAT puisse transmettre des informations dans le cas de tests d'équipements, et ainsi

l'administration fédérale d'aviation FAA (`Federal Aviation Administration`), pour l' Europe, il s'agit de l'agence européenne de la sécurité aérienne EASA (`European Aviation Safety Agency`).

potentiellement propager une attaque à bord de l'avion, du fait qu'il soit considéré comme un équipement sol, il n'existe à ce jour aucune obligation de certification pour ce type d'équipement.

Dans la suite de cette section, nous présentons un premier scénario de maintenance tel qu'il est réalisé actuellement, puis nous décrivons un deuxième scénario utilisant un support électronique(le PMAT).

II.2 Scénario de maintenance et son évolution

Afin d'optimiser la disponibilité des appareils et augmenter leur rentabilité, les opérations de maintenance fréquentes sont mises en œuvre entre les phases de vol. Ces opérations ont un impact significatif sur le temps d'escale de l'appareil. Effectivement, un vol peut être retardé si les opérations de maintenance prévues ne sont pas terminées ou si un équipement est défectueux. Le bilan de l'année 2010 [Cabanes & Petit 2010] réalisé par l'observatoire des retards du transport aérien note que 29,4% de vols ont été retardés. Ces retards sont imputables pour 28% aux compagnies qui représentent la deuxième cause de retard après l'enchainement des rotations [2]. L'analyse des retards imputés à la compagnie montre que 14% des ces retards sont dus à un problème de maintenance. Cela signifie que 4,14% des vols de l'année 2010 ont été retardés à cause d'un problème de maintenance.

Les opérations de maintenance concernent aussi bien l'extérieur de l'appareil (trace d'usures du fuselage, anomalie de la structure) que l'intérieur (équipement de bord, écran vidéo des passagers, etc.). Les opérations de maintenance « extérieures » nécessitent une intervention directe de l'équipe de maintenance. Il est hors de notre étude de proposer une optimisation permettant de réduire le temps de cette intervention. Nos travaux se concentrent sur les interventions au niveau des équipements de bord. Nous détaillons dans la suite de cette section un scénario actuel de maintenance avant de proposer une possible évolution.

Les scénarios de maintenance qui vont être présentés dans les sections suivantes ont pour objectif de décrire le cas d'étude utilisé pour nos travaux. De ce fait, les scénarios reflètent uniquement les grandes lignes des opérations de maintenances.

II.2.1 Maintenance classique

Afin de contrôler le comportement des systèmes à bord de l'appareil, chaque système inclut le BITE (*Built in Test Equipment*) utilisé pour la détection des équipements défectueux. À cela s'ajoute des sondes spécifiques pour recueillir des informations qui sont ensuite transmises à un ordinateur central de maintenance. L'opérateur de maintenance peut consulter les informations collectées à travers différents moyens à bord de l'avion (imprimante, MCDU (*Multipurpose Control and Display Unit*) et MDDU (*Multi Disk Drive Unit*) ou en envoyant l'information par le système ACARS (*Aircraft Communication Adressing and Reporting System*) directement à la compagnie aérienne propriétaire de l'appareil. Ces informations peuvent être par exemple des détails relatifs à chaque test

2. Une rotation est une suite de vols débutant et se terminant à la base d'affectation de l'appareil.

effectué : nature de l'équipement, nature du test, nature de la faute identifiée, code de la faute, code de la procédure de traitement de la faute, etc. L'ensemble des équipements relatifs à la maintenance fait partie de l'OMS (*On-board Maintenance System*)

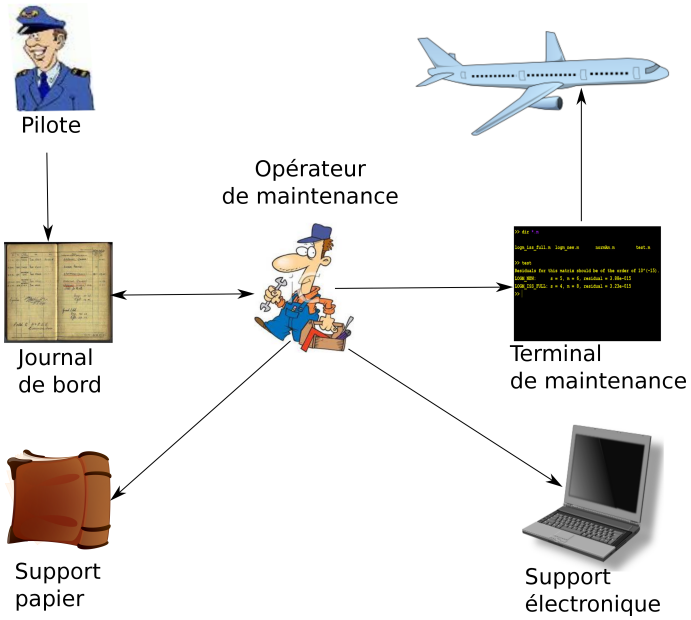

Figure II.1 – Scénario classique de maintenance

Ainsi, les équipements envoient périodiquement des informations relatives à leur état interne de fonctionnement durant la phase de vol. Tout dysfonctionnement est enregistré dans un journal de bord (cf. Figure II.1). Le journal de bord est ensuite analysé durant la phase de maintenance et les incidents sont classés selon leur gravité, afin d'optimiser le temps de traitement par les opérateurs de maintenance.

Actuellement, durant les opérations de maintenance, l'opérateur analyse les différents rapports (journal de bord, rapport d'équipements, etc.) en s'appuyant sur des manuels de maintenance. Ces manuels sont généralement sous forme de documents papiers, que l'agent de maintenance doit avoir avec lui lors de sa mission. Ces documents sont progressivement mis sous format électronique, permettant ainsi à l'agent de les consulter directement sur un ordinateur portable. Cependant, cet ordinateur n'est pas connecté au réseau de l'avion. Il sert uniquement de support électronique de consultation des manuels, comme nous le montre la Figure II.1. Cet ordinateur pourrait être apparenté à un EFB de classe 1.

Suite à la consultation de la procédure de maintenance sur son ordinateur de maintenance, l'opérateur lance la batterie de tests qui est demandée dans cette procédure, par le biais d'un équipement dédié à bord de l'avion : le terminal de maintenance, c'est notamment le cas pour l'A380. La présence d'un opérateur humain durant la phase de maintenance est primordiale pour la prise de certaines décisions. Dans le cas ou un calculateur est signalé comme défaillant, l'opérateur a le choix de retarder le décollage pour permettre le remplacement du calculateur, ou bien suivre une procédure en mode dégradé sous certaine condition, par exemple que ce calculateur soit remplacé lors de la prochaine escale.

Ce type de problème place l'opérateur au cœur de la phase de maintenance, il doit

en fonction des erreurs signalées, choisir l'action a accomplir afin d'y remédier, tout en prenant en compte des contraintes extérieures notamment des contraintes économiques. De plus, le fait de devoir utiliser un terminal unique à bord de l'avion représente une perte de temps importante pour les opérations de maintenance aussi bien à l'intérieur qu'à l'extérieur de l'avion.

Nous nous plaçons dans ce cadre opérationnel et proposons d'optimiser l'opération de maintenance en la mettant en œuvre sur un support électronique unique, le PMAT, tout en permettant à l'agent de maintenance de contrôler le bon déroulement de ces opérations, en se basant sur son expérience, et sur des tests intermédiaires qu'il pourra lancer sur les équipements de l'avion. De plus, l'utilisation d'un ordinateur portable augmente la mobilité de l'opérateur, ce qui peut représenter un gain de temps significatif pour la compagnie.

Dans la section suivante, nous proposons de détailler le scénario fonctionnel d'une telle opération de maintenance.

II.2.2 Maintenance avec support électronique

Dans le cadre des opérations classiques de maintenance, l'opérateur est contraint de servir d'interface de communication entre ses manuels électroniques (contenant en particulier les procédures à suivre pour tester un équipement) et le terminal de maintenance sur lequel il effectue les actions dictées par ces procédures. Ce mode de fonctionnement permet d'éviter la connexion directe de l'ordinateur portable à l'avion, annulant ainsi tout risque de remontée de flux d'information depuis ce portable vers des équipements critiques.

La Figure II.2, présente une autre approche de la maintenance.

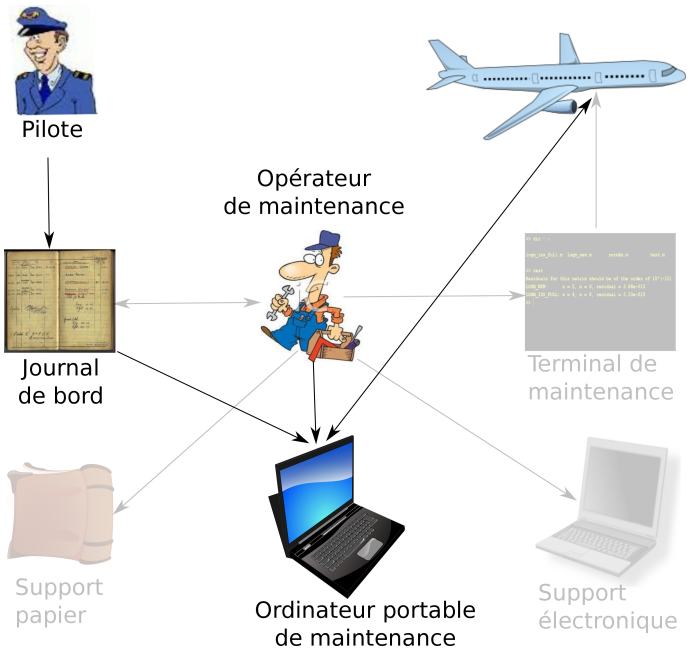

Figure II.2 – Scénario de maintenance avec support électronique

Dans cette approche, les opérations de maintenance bénéficient d'une meilleure inté-

gration des moyens électroniques. En effet, les manuels de maintenance (aujourd'hui sous forme électronique ou papier) décrivant les procédures à accomplir sont pré-enregistrés sur l'ordinateur de maintenance. Lorsqu'il est connecté à l'avion, l'ordinateur de maintenance récupère les rapports d'anomalies provenant du journal de bord, les classifie et indique pour chaque anomalie la nature de la faute qui lui correspond, et les actions à réaliser pour la traiter. Dans chaque procédure, des liens vers des applications de maintenance spécifiques sont fournis, permettant à l'opérateur de lancer directement les tests et opérations nécessaires pour réaliser ces procédures.

Remarquons que dans ce cas, l'opérateur humain n'a plus à définir chacune des actions exigées par la procédure, mais il se contente de suivre la procédure, de vérifier la validité des informations qui s'affichent et de veiller au bon déroulement des actions en cours. Dans certains cas, il peut être demandé à l'opérateur de se déplacer dans l'appareil et de réaliser lui-même certaines actions sur les équipements. Par exemple, dans certaines procédures où les volets de l'avion sont actionnés, il est demandé à l'opérateur de se déplacer pour vérifier que les volets ont bel et bien été actionnés, puis de valider cela au niveau de l'ordinateur de maintenance. Ceci est facilité si la connexion entre l'ordinateur portable de maintenance et l'OMS est une liaison sans fil. Une telle assistance à l'opérateur dans les opérations de maintenance conduit à un gain de temps considérable, rendant ainsi l'appareil opérationnel plus rapidement. Cette propriété fait l'objet d'une forte demande et est fort appréciée par les compagnies aériennes qui cherchent à réduire au maximum le temps d'escale. En effet, dans la plupart des aéroports, la compagnie doit s'acquitter des frais supplémentaires en cas de dépassement de la plage horaire qui lui est assignée. De plus, l'image de la compagnie est altérée lors d'un dépassement de ces plages, puisque les retards induits conduisent à une mauvaise appréciation de la part des clients.

La modification du poste de travail de l'opérateur de maintenance en allégeant les tâches répétitives permet également de limiter les erreurs pouvant survenir suite à une faute d'interaction. En effet, la forte pression que peuvent exercer les compagnies afin de minimiser les retards et assurer la continuité du service de l'appareil joue un rôle significatif dans la fréquence d'apparition de telles fautes.

Notons que l'ordinateur portable utilisé pour les opérations de maintenance à bord de l'avion peut également servir afin de rédiger des rapports d'analyse, consulter la mise à jour des procédures sur le site internet du constructeur ou tout autre tâche administrative lorsque l'opérateur de maintenance se trouve dans les locaux de la compagnies aérienne ou dans ceux du sous-traitant en charge de la maintenance. Aussi, il apparait clairement un problème de sécurité des équipements de bord lorsqu'un même ordinateur portable est utilisé dans un environnement de très forte criticité (connexion à des équipements à bord de l'avion) et dans un environnement peu critique (connexion au site du constructeur).

Dans la suite de cette section nous décrivons les différents problèmes liés à l'utilisation d'un ordinateur portable.

II.3 Problème soulevé

L'utilisation d'un ordinateur de maintenance en connexion directe avec l'avion pose un problème de sécurité-immunité pour les raisons suivantes. Tout d'abord, l'ordinateur,

de type portable, est fourni par la compagnie et peut être considéré comme un ordinateur personnel de l'agent de maintenance. Il peut donc être déplacé et connecté à des réseaux du monde ouvert, et potentiellement être connecté à l'Internet, par exemple pour du courrier électronique ou de la recherche d'informations sur le Web. Cette hypothèse de fonctionnement expose le système d'exploitation ainsi que les applications qui y sont installées à des attaques (des vers, des chevaux de Troie, etc.), diminuant de la sorte le degré de confiance que l'on peut accorder aux sorties de cet ordinateur.

Pourtant, cet ordinateur de maintenance doit pouvoir être connecté à l'avion afin de réaliser des tests et éventuellement procéder à des remplacements d'entités logicielles à bord. Comme nous l'avons mentionné dans le Chapitre I sous-section I.3.3, les niveaux de criticité dépendent uniquement de la nature de la tâche accomplie. C'est aux concepteurs et développeurs de cette tâche sur un module spécifique de prouver que le niveau de confiance obtenu pour ce module est au moins égal à celui de la criticité de la tâche.

Dans chaque niveau de criticité, une ou plusieurs tâches sont implantées. Elles peuvent donc communiquer entre elles. Par contre, une tâche de faible niveau de criticité peut être amenée à communiquer avec une tâche de plus haut niveau de criticité. Cette situation est illustré dans la Figure II.3 où les flèches représentent les flux logiques qui peuvent exister entre différentes tâches.

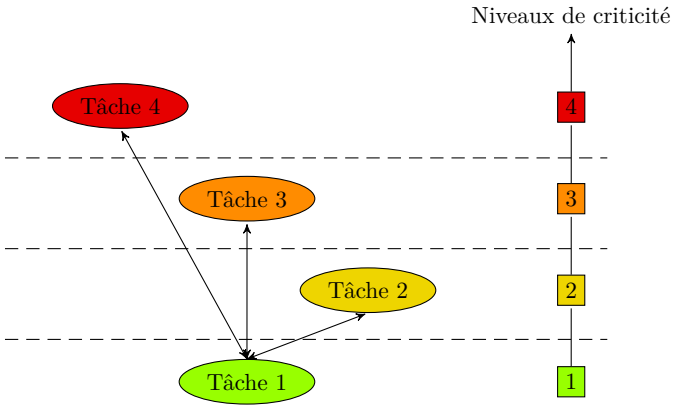

Figure II.3 – Niveaux de criticité de tâches

Ainsi, la tâche 1 (de niveau de criticité 1) qui correspond ici à l'application installée sur l'ordinateur de maintenance, peut envoyer des informations vers des tâches de niveaux de criticité différent (la tâche 2 de niveau 2, la tâche 3 de niveau 3 et la tâche 4 de niveau 4). Ces tâches peuvent être associées à des applications du système avionique ayant des niveaux de criticités différents.

Un contrôle de flux doit être mis en place afin de permettre de telles communications tout en évitant que la tâche 1 ne dégrade le fonctionnement des autres tâches. En effet, à défaut de tels mécanismes de protection, une attaque réalisée au niveau de l'ordinateur de maintenance serait susceptible de corrompre une tâche du domaine des systèmes de contrôle de l'appareil à travers une corruption intermédiaire du domaine des systèmes d'opération et de maintenance de l'appareil.

Dans la section suivante, nous décrivons l'architecture mise en place notamment pour contrôler les flux d'informations.

II.4 Architecture logicielle de l'équipement

Afin de pallier le problème de communication entre entités de niveaux de criticité différents, nous devons nous assurer de l'intégrité de l'information échangée. L'utilisation d'un modèle de communication permettant la cohabitation d'entités logicielles de niveaux de criticité différents dans le domaine avionique a été étudié dans [Laarouchi 2009]. Nous reprenons la description du modèle de communication présenté dans le Chapitre I Section I.3.4 et nous présentons l'architecture qui a été mise en place en utilisant ce modèle.

II.4.1 Modèle Totel et redondance

Le modèle Totel (cf. Chapitre I Section I.3.4) permet, sous certaines conditions, une remontée de flux d'information d'un niveau de faible criticité vers un niveau plus critique. L'autorisation d'un telle communication est assurée par un objet de validation qui utilise différentes techniques de tolérance aux fautes que nous détaillons par la suite.

Deux techniques sont principalement décrites par [Laarouchi 2009] : le contrôle de vraisemblance et le contrôle par redondance. Dans la suite de cette section, nous détaillons ces différentes techniques.

Les **contrôles de vraisemblance** consistent à appliquer des assertions sur les données provenant de niveaux inférieurs de confiance (par exemple, vérifier la validité des données par rapport à un intervalle de confiance) afin de se prononcer sur l'acceptabilité des données.

Les contrôles de vraisemblance présentent l'avantage de n'entrainer qu'un surcoût peu important par rapport aux éléments fonctionnels du système. Ils permettent de faire face aux erreurs induites par un large spectre de fautes, autant accidentelles que malveillantes. Les contrôles de vraisemblance peuvent être mis en œuvre par :

- du matériel spécifique capable de détecter des valeurs erronées (violation de code détecteur d'erreur, adresse mémoire inexistante, instruction inexistante), ou des violations de protections de segments de mémoires ;
- du logiciel spécifique capable de vérifier la conformité des entrées ou sorties du système par rapport à un invariant.

Les contrôles de vraisemblance mis en œuvre par logiciel peuvent être intégrés au système d'exploitation et être ainsi disponibles pour toutes les applications du système ou spécifique à une sémantique particulière (intervalle de valeur possible, écart maximal par rapport au résultat précédent dans un contrôle de processus continu, etc.). Pour ce dernier prenons l'exemple d'un système de capteur en charge de contrôler la température d'un four dans une usine métallurgique. Un contrôle de vraisemblance peut être la vérification que la mesure de la température ne soit pas négative ou supérieure à 1000°Celsius (en supposant que 1000°est la température maximum de fonctionnement du four). Un tel contrôle de vraisemblance est capable de détecter si le capteur est en dehors de l'intervalle défini mais ne peut pas détecter un intervalle d'erreur, par exemple lorsque le capteur affiche 95°alors que la température réelle est de 105°. Si ce type d'erreur est critique, il peut être essentiel d'utiliser une seconde technique de tolérance aux fautes tel que le contrôle par redondance.

La **redondance** est basée sur l'utilisation de multiples sources d'informations et de mécanismes de comparaison et de décision (par l'intermédiaire de l'OV du modèle Totel) afin de décider quelles données peuvent être transmises au niveau supérieur.

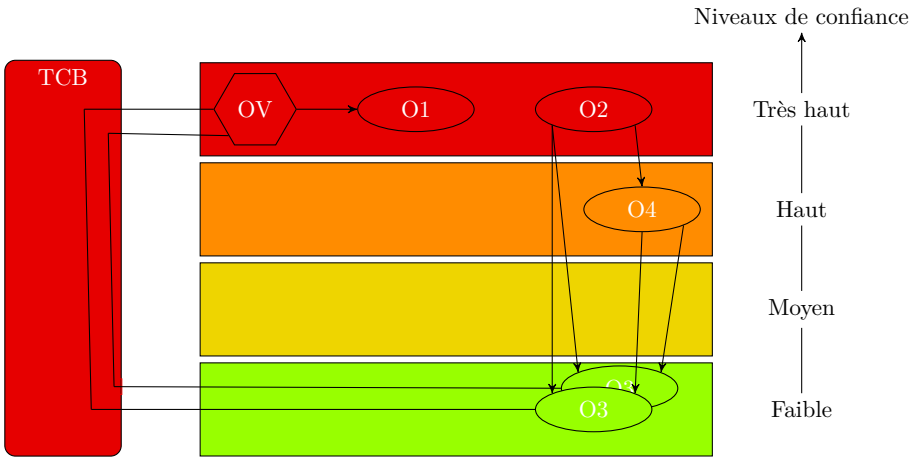

Figure II.4 – Modèle Totel : OV utilisé comme outil de décision par redondance d'information

Par exemple, l'OV représenté dans la Figure II.4 compare des sources d'informations multiples (dans cette figure l'objet O3 est composé de plusieurs copies) et prend une décision sur la donnée à transmettre à l'objet O1. Cette technique suppose que des sources différentes livrent des données redondantes contenant les mêmes informations.

Il est important de spécifier un niveau cohérent de comparaison pour que les éléments comparés soient assez expressifs pour éviter des fausse alertes (c'est-à-dire, des faux-positifs). Par exemple, si nous considérons deux fichiers JPEG, une différence d'un pixel entre les deux fichiers ne signifie pas nécessairement que les deux images soient perçues comme distinctes par un utilisateur humain. Par conséquent, dans ce contexte, l'utilisation des pixels comme niveau de comparaison ne serait pas un choix judicieux, puisque il pourrait amener à des fausses alertes.

L'OV doit être conçu et développé en respectant les hypothèses de faute pour que le résultat produit soit digne de confiance. Il est impératif que les défaillances des sources redondantes d'informations soient indépendants. Cela influence le type de redondance qui peut être déployé en respectant les types de fautes à tolérer. Par exemple, pour tolérer des fautes physiques sur le matériel, les copies identiques de même source peuvent être utilisées. Cependant, si nous considérons des fautes malveillantes ou des défauts de conception qui induisent des sources identiques, cet état de fait est clairement insuffisant. L'utilisation de versions distinctes en diversifiant les sources est nécessaire pour la protection efficace contre ces classes de fautes.

Cette technique suppose aussi que les éléments comparés représentent les mêmes informations, présentées dans le même ordre, ce qui conduit à une exigence de déterminisme des répliques. Une telle exigence n'est pas toujours facile à satisfaire. Ainsi les auteurs de [Poledna 1994] décrivent de nombreuses causes potentielles de non-déterminisme (par exemple le *multithreading*, ou des *timeouts*) qui pourraient amener des sources d'information exemptes de fautes à se comporter différemment et par conséquent transmettre des données pouvant être détectées comme différentes lorsqu'elles sont comparées par l'OV.

Nos travaux se focalisent sur les fautes malveillantes, particulièrement celles qui ciblent le système d'exploitation. Par conséquent, nous favorisons la diversification comme une technique de tolérance aux fautes permettant à l'OV de tolérer des attaques. Dans la suite de cette section nous présentons l'architecture mise en place afin d'autoriser une communication bidirectionnelle entre entités redondantes.

II.4.2 Présentation de l'architecture logicielle

Nous rappelons que l'architecture présentée dans cette section à pour objectif d'être déployée sur ordinateur portable de type COTS afin d'accomplir les différentes actions de maintenance que nous avons présentées à la Section II.2. Les différentes techniques de virtualisation, que nous présenterons plus en détail au Chapitre IV, peuvent être utilisées afin de réaliser aussi bien des fonctions d'isolation que de détection ou de recouvrement d'erreurs. Dans la suite de cette section, nous détaillerons les différentes couches logicielles utilisées.

La première couche, qui s'exécute directement sur le matériel est un moniteur de machine virtuelle ou plus communément appelé hyperviseur. Nous avons précédemment indiqué que l'hyperviseur pouvait assurer aussi bien l'isolation que la détection et le recouvrement. Cependant, dans l'architecture que nous proposons l'hyperviseur assure uniquement l'isolation. En effet, les tâches de détection et de recouvrement sont déléguées à l'objet de validation (OV). Ainsi, le contrôle de la validité des deux instances est réalisé au niveau de l'OV qui a pour rôle d'implémenter le mécanisme de décision approprié afin de valider les résultats.

Diverses possibilités d'implantations de l'OV peuvent être envisagées. Rappelons que l'objet de validation doit avoir le même niveau d'intégrité que l'objet dont il valide les données. Le support d'exécution de l'objet doit être de confiance afin d'éviter qu'une entité sûre ne s'exécute sur une plateforme non sûre. Dans ce qui suit, nous présentons trois niveaux où il est possible d'implémenter l'objet de validation : au niveau de l'hyperviseur, du code binaire, d'une machine virtuelle sûre.

Emplacement de l'objet de validation

Implémentation au niveau de l'hyperviseur
Une première possibilité est d'implémenter l'objet de validation au sein de l'hyperviseur afin de procéder à la comparaison des flux de données. Rappelons qu'un flux consiste en un envoi de données depuis un objet du système vers un objet d'un autre système. Les objets étant des applications logicielles, les flux seront identifiés comme tout échange de données entre les applications des systèmes. En effet, l'hyperviseur est une entité sûre de fonctionnement qui respecte la condition que nous avons citée ci-dessus. Cependant, l'implantation d'une nouvelle fonctionnalité dans l'hyperviseur doit être du même niveau de confiance (par conséquent développée et validée par des méthodes adéquates). De plus, il est nécessaire de s'assurer qu'une telle implantation ne modifie pas le comportement de cet hyperviseur. Le travail à fournir peut s'avérer long et coûteux. En effet, pour faciliter leur certification, les hyperviseurs doivent être minimaux et ne disposer que de primitives basiques. L'intégration d'un objet de validation complexe nécessiterait de mettre en œuvre

des méthodes de développement et de certification spécifiques.

Implémentation au niveau d'un binaire
Certains hyperviseurs permettent d'exécuter directement du code binaire, sans avoir à passer par un système d'exploitation invité. Ce niveau d'implémentation ne modifie pas l'hyperviseur. Par contre, l'objet de validation doit être en mesure de dialoguer avec toutes les bibliothèques n'existant pas dans l'hyperviseur et dont l'objet de validation aurait besoin pour des fonctionnalités spécifiques. Naturellement, de telles bibliothèques doivent être validées au niveau d'intégrité de l'objet de validation. Une telle tâche peut également s'avérer coûteuse.

Implémentation au niveau d'une machine virtuelle sûre
L'implémentation au niveau d'une machine virtuelle sûre, qui dispose d'un niveau de confiance approprié, permet à l'objet de validation d'utiliser toutes les bibliothèques du système invité, sans avoir à les revalider ni les développer comme c'est le cas des options précédentes. Une telle implantation ne modifie ni l'hyperviseur ni le système invité. En effet, l'objet de validation est implémenté sous la forme d'une application qui s'exécute sur le système sûr. Ce dernier, de par sa nature, permet une telle exécution tout en assurant son intégrité. Notons également que de tels systèmes d'exploitation existent et sont déjà utilisés en avionique, il serait donc possible d'en tirer profit en les intégrant dans notre architecture globale.

Bien que cette dernière option soit la plus simple à mettre en œuvre, demandant le moins de modification et, toutefois, comme nous n'avons pu obtenir un système d'exploitation sûr, nous avons utilisé l'implémentation de l'objet de validation au niveau de l'hyperviseur dans le démonstrateur que nous présenterons au Chapitre V. La Figure II.5 présente l'architecture retenue. L'OV est effectivement implémenté dans l'hyperviseur. De plus, deux machines virtuelles différentes sont présentées avec deux systèmes d'exploitation différents. Par contre, la tâche exécuté par les deux systèmes d'exploitation est la même.

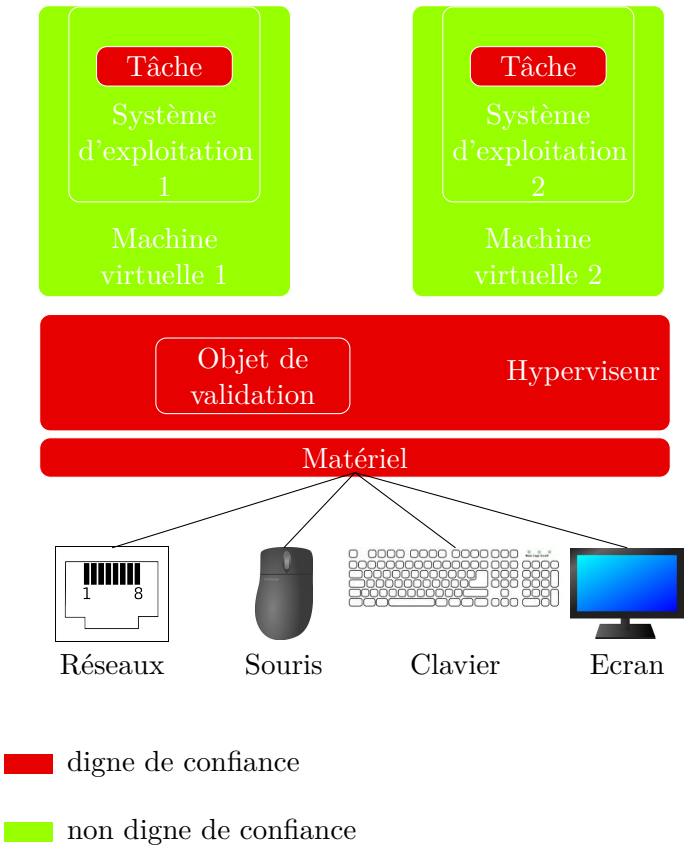

Figure II.5 – Architecture logicielle de l'équipement

Mécanisme de comparaison

Maintenant que nous avons défini l'implémentation de l'objet de validation ainsi que les principales couches logicielles de l'architecture, nous décrivons le mécanisme de comparaison des données utilisé par l'objet de validation après avoir présenté les deux tâches représentées sur la Figure II.5.

Ces deux tâches sont des instances d'une application, ce qui signifie qu'elles remplissent exactement les mêmes fonctions. Comme nous l'avons précisé dans la Section II.2, l'application utilisée est consacrée aux différentes opérations de maintenance. Pour des raisons de portabilité l'application est développée en langage Java. En effet, une application Java n'est pas exécutée directement par le système d'exploitation, mais par un interpréteur connu sous le nom de JVM, pour *Java Virtual Machine*, dont les spécifications sont données par [Lindholm & Yellin 1999]. Il est donc possible d'exécuter une même application sur différents systèmes d'exploitation (par exemple *Windows* et *Linux*).

Les interactions, réalisées par les différentes instances de l'application, doivent être comparées les unes aux autres. Elles doivent donc être interceptées afin d'être envoyées à l'objet de validation. Dans la suite, nous présentons les différentes manières d'intercepter ces interactions.

Interception des événements de l'environnement d'exécution
Lorsque l'application Java interagit avec son environnement, en particulier à travers le réseau ou avec l'écran, le clavier ou la souris, un message est envoyé à travers le système d'exploitation au composant ciblé par l'interaction. Par conséquent, nous pourrions capturer ces événements directement en modifiant le système d'exploitation ou en instrumentant la machine virtuelle invitée. Dans les deux cas, nous sommes face à deux problèmes difficiles à résoudre. Le premier problème concerne les modifications à apporter pour mettre en place les interceptions. Pour les mettre en place, il est nécessaire de modifier les machines virtuelles invitées ou les systèmes d'exploitation. Ces modifications sont dépendantes du type de système d'exploitation. La mise à jour d'un des systèmes d'exploitation ou l'intégration d'un nouveau système d'exploitation nécessite alors de modifier le mécanisme d'interception.

Le deuxième problème concerne la granularité de l'information à comparer. La création d'un événement au niveau de l'application Java (par exemple, l'envoi d'un message Java à travers le réseau) se traduit par une succession d'événements au niveau du système d'exploitation (par exemple, ouverture d'une connexion réseau avec poignée de main, échange de messages liés au protocole de communication, échange des données encodées, etc.). Or, de nombreuses informations (les métadonnées) deviennent dépendantes du contexte d'exécution (date de la création de l'événement, empreinte de la machine virtuelle invitée, type de système d'exploitation, etc.) et il est difficile de faire la part entre les données issues de l'application et ces métadonnées. S'il n'est pas possible d'être suffisamment précis dans cette identification des données, nous risquons de laisser circuler des informations différentes d'une instance de l'application à l'autre.

À l'instar de l'instrumentation du système d'exploitation, l'instrumentation de la JVM pour la comparaison des messages, pose des problèmes liés à la granularité des informations comparées et la flexibilité d'intégration de nouvelles versions des JVMs.

Afin d'effectuer les comparaisons, il serait possible de modifier directement le code source de l'application afin d'intégrer les requêtes permettant de comparer les interactions. Dans ce cas, les modifications doivent être apportées par le concepteur de l'application Java lui-même. Elles sont similaires à celles présentées dans la Figure II.6. Cette figure illustre le principe sur du code source, mais le principe est équivalent sur le bytecode Java. Plus précisément, deux injections de code sont nécessaires : une avant l'invocation pour permettre d'effectuer une comparaison avant chaque entrée/sortie et une après l'invocation pour tester le retour.

```
void afficher_message(String message)
{
        label.setText(message);
 }
```

(a) code source non modifié

```
void afficher_message(String message)
{
        //hyperviseur.comparer(message)
        retour=label.setText(message);
        //hyperviseur.comparer(retour)
 }
```

(b) code source modifié par injections de code

Figure II.6 – Instrumentation par modification du code source de l'application

Cette approche présente l'inconvénient de ne pas être transparente pour le développeur de l'application. Ce dernier doit savoir si l'application sera exécuté dans un environnement

diversifié. Cette approche n'est donc pas souhaitable.

L'instrumentation du code binaire de l'application repose sur la modification à la volée du code de l'application chargée par le *ClassLoader* de la machine virtuelle. Plus précisément, lorsque le code Java en cours d'exécution correspond à l'accès à une classe, le *ClassLoader* est invoqué. Ce dernier vérifie tout d'abord si la classe en question a déjà été chargée. Si c'est le cas, la méthode de la classe est invoquée. Sinon, en fonction du nom de la classe, le *ClassLoader* accède au système de fichier et ouvre en lecture le fichier contenant le code de la classe. Ce code est chargé et vérifié (intégrité, contrôle d'accès, etc.). L'approche présentée dans cette section consiste à remplacer le *ClassLoader* par défaut, qui n'est autre qu'une classe Java, par un *ClassLoader* qui va analyser le code chargé pour injecter des instructions. Cette approche peut être mise en place par l'utilisation de la programmation orientée aspect [Kiczales *et al.* 1997], en particulier AspectJ [Kiczales *et al.* 2001] ou bien par l'utilisation de bibliothèque telle que ASM [Bruneton *et al.* 2002]. Un exemple d'utilisation de la bibliothèque ASM est donné dans [Kuleshov 2007].

La notion de *ClassLoader* est standard dans la spécification des JVMs. Par conséquent, cette approche est portable sur toutes les JVMs utilisées dans l'architecture. Ensuite, l'injection étant réalisée directement dans le code de l'application Java, les informations capturées peuvent être comparées directement entre les instances sans avoir à les traiter au niveau sémantique. C'est cette approche que nous employons pour réaliser la comparaison.

Lors du démarrage de la JVM, les classes de la bibliothèque standard sont chargées directement avant la classe correspondant au *ClassLoader*. Le mécanisme d'injection présenté précédemment, ne fonctionne donc que sur les classes de l'application et non sur les classes standard. Par conséquent, nous ne pouvons pas injecter directement du code dans les constructeurs et les méthodes des classes permettant de réaliser des entrées/sorties. Il est donc nécessaire de mettre en place une stratégie nous assurant de capturer toutes ces invocations d'une autre manière. Cette stratégie est la suivante :

1. Injection de code dans le code du constructeur d'une classe qui hérite directement d'une classe d'entrée/sortie.
2. Injection de code autour de l'invocation des méthodes non surchargées d'une classe d'entrée/sortie.
3. Injection de code autour de l'invocation de l'instanciation d'une classe d'entrée/sortie.
4. Injection de code au début des méthodes susceptibles d'être invoquées directement par la bibliothèque Java (méthodes nommées *callback*, correspondant, par exemple, à la méthode invoquée lors de l'action sur un bouton).
5. Injection de code dans le constructeur d'une classe qui n'hérite pas d'une classe d'entrée/sortie, qui n'implémente pas indirectement une interface d'entrée/sortie mais qui implémente directement une interface d'entrée/sortie.
6. Injection autour de la méthode *run* d'une classe de type *Thread* ou *Runnable*.

L'interception des méthodes présentées précédemment, aboutit à l'envoi d'un message depuis les machines virtuelles vers l'objet de validation. Ce message contient les différents champs permettant d'effectuer la comparaison :

- l'identifiant du *thread* en cours d'exécution ;
- le numéro de message unique ;
- le nom de la méthode invoquée ;

- la liste des paramètres, les objets sont remplacés par leurs références.

Un chien de garde est utilisé pour définir le délai de transmission entre les machines virtuelles et l'objet de validation. Les messages des différentes instances de l'application doivent être sémantiquement identiques [3] et respecter le délai imposé par le chien de garde pour être considérés comme identiques.

Suite à la comparaison, l'OV prend la décision, si les traces sont déclarées équivalentes, de transmettre le message au destinataire. Dans le cas contraire, lorsque le chien de garde est déclenché, ou bien lorsque les messages ne sont pas déclarés équivalentes, l'OV ne transmet pas le message au destinataire.

II.4.3 Utilisation des traces d'exécution pour comparaison

Les traces d'exécution de l'application de maintenance concernent les appels de méthodes graphiques. Le Tableau II.1 présente un exemple de trace d'exécution composée de quatre messages.

$Trace_1$	Message number	Method name	Parameter
new Button()	1	Button	null
new Button()	2	Button	null
setValue(12)	3	setValue	12
setVisible(True)	4	setVisible	True

Tableau II.1 – Exemple de trace

Les Tableaux II.2 et II.3 décrivent des exemples de traces issue de deux machines virtuelles. Les traces numéro 1 sont identiques sur les deux machines virtuelles, et sont par conséquent déclarées comme équivalentes. Les traces numéro 2 ne sont pas identiques à cause de la valeur placée en paramètre de la méthode `setValue`. Les traces numéro 3 ne sont différentes que par le nom la méthode invoquée.

$Trace_1$	$Trace_2$	$Trace_3$
new Bouton()	new Bouton()	new Bouton()
new Bouton()	new Bouton()	new Bouton()
setValue(12)	setVisible(True)	`setVisible(True)`
setVisible(True)	`setValue(12)`	setValue(12)

Tableau II.2 – Traces d'exécution issue de la machine virtuelle 1

II.5 Proposition d'amélioration de l'architecture logicielle

L'architecture logicielle du PMAT utilise la technique de virtualisation afin de diversifier l'environnement d'exécution de l'application de maintenance. Dans ce contexte, comme

3. les messages doivent avoir les mêmes numéros, invoqués les mêmes noms de méthodes avec les mêmes paramètres.

$Trace_1$	$Trace_2$	$Trace_3$
new Bouton()	new Bouton()	new Bouton()
new Bouton()	new Bouton()	new Bouton()
setValue(12)	setVisible(True)	`setValue(12)`
setVisible(True)	`setValue(666)`	setVisible(True)

Tableau II.3 – Traces d'exécution issue de la machine virtuelle 2

nous venons de le décrire, le mécanisme de comparaison exécuté par l'OV utilise un chien de garde afin de borner le délai de la transmission de l'information entre les machines virtuelles et l'OV. Nous rappelons que l'OV est implémenté au niveau de l'hyperviseur. Ce mécanisme, bien que performant et de faible coût, nécessite une grande précision dans la définition du délai après lequel il doit alerter le système. En effet, prenons l'exemple d'un simple chien de garde en charge de détecter si une application ne donne pas de réponse au bout d'un délai d préalablement fixé. Il est clair que d dépend aussi bien de l'application que de son environnement. Ainsi, si la valeur de d n'est pas choisie en prenant en compte ces paramètres internes et externes, le chien de garde risque soit de générer beaucoup de fausses alarmes (ce qui est potentiellement bloquant pour le reste du système), soit de ne pas générer d'alarmes assez rapidement alors que l'application est défaillante (ce qui risque d'avoir des conséquences encore plus catastrophiques sur le système).

Afin que le mécanisme de comparaison ne repose pas uniquement sur l'utilisation d'un chien de garde, nous proposons d'ajouter au mécanisme précédemment décrit, un deuxième mécanisme de comparaison. L'utilisation de deux processus permettra d'augmenter la couverture de détection. Ce deuxième mécanisme, basé sur l'utilisation d'un modèle d'exécution de l'application sera détaillé au Chapitre III.

Une deuxième amélioration consiste à s'assurer que la couche de virtualisation ne soit pas à l'origine d'une détection à tort par manque de performance. En effet, la couche de virtualisation et plus précisément l'hyperviseur est au centre de notre architecture. De plus, du fait que l'OV soit implémenté dans l'hyperviseur, la question des performances de l'hyperviseur doit être posée. Une étude doit donc être menée afin de définir l'impact de son utilisation sur les performances du système. Pour ce faire, nous avons défini et réalisé des tests de performance sur un hyperviseur que nous détaillerons dans le Chapitre IV.

Conclusion

Nous avons débuté ce chapitre par la présentation d'équipements mobiles utilisés dans le domaine avionique. En nous appuyant sur l'un d'entre eux, le PMAT, nous avons décrit un scénario et proposé une possible évolution de son utilisation. Nous avons considéré ce dernier scénario comme cas d'étude et présenté l'architecture logicielle à déployer sur le PMAT afin de réaliser de telles opérations. Le scénario montre la nécessité de remontée de flux d'un niveau peu critique vers un niveau plus critique. Dans un dernier temps, nous avons décrit les possibles améliorations à apporter à l'architecture.

Notre proposition d'architecture utilise des techniques de virtualisation afin d'assurer la diversification de système d'exploitation. Un premier mécanisme de comparaison a été présenté à la Section II.4.2. L'utilisation d'un chien de garde a été présenté et nous avons montré la précision à apporter à un tel mécanisme afin de ne pas perturber inutilement le système. Nous proposons afin d'augmenter la couverture de détection, l'ajout d'un deuxième mécanisme de comparaison basé sur l'utilisation d'un modèle d'exécution de l'application . Cette démarche est décrite dans le Chapitre III.

La virtualisation est à la base de cette architecture logicielle, il est donc primordial de s'assurer que son utilisation n'entraine pas des détections à tort par manque de performance. Pour cela, nous réalisons des tests de performances sur plusieurs hyperviseurs que nous détaillons dans le Chapitre IV.

Chapitre III

Détection d'erreurs par analyse comportementale

Introduction

Dans l'implémentation présentée à la section II.4.3 du Chapitre II, les attaques sont détectées en comparant les traces d'exécution provenant des deux instances de l'application. L'hypothèse se fonde sur le fait qu'une différence entre les traces d'exécution est considérée comme le résultat d'une attaque. De façon à empêcher l'attaque de progresser vers le système d'information de l'avion, notre architecture est conçue de manière à arrêter l'exécution de l'application.

Cependant, l'absence de différence entre les traces d'exécution n'implique pas nécessairement l'absence d'une attaque. La détection d'attaque basée seulement sur la comparaison de traces d'exécution peut avoir une couverture incomplète. En particulier, les fautes de mode commun ne sont pas prise en compte.

Dans ce chapitre, nous proposons [Lastera *et al.* 2012] de renforcer le processus de détection d'attaque, en y ajoutant un critère de décision s'appuyant sur un modèle d'exécution de l'application afin d'accroître la couverture de détection. Dans un premier temps, nous détaillons le principe de la démarche. Ensuite, nous décrivons le processus de construction du modèle d'exécution de l'application avant de présenter, dans un troisième temps, la mise en place de cette approche dans le cycle de vie de l'application. Nous terminons ce chapitre en montrant la complémentarité des mécanismes utilisés afin de détecter une attaque.

III.1 Axe de développement de l'architecture

Nous souhaitons augmenter la couverture de détection de notre architecture afin d'améliorer le mécanisme existant. Nous rappelons que le mécanisme de détection par comparaison

repose sur l'hypothèse qu'une attaque sur un système d'exploitation n'aura pas les mêmes effets sur des systèmes d'exploitation différents. Cependant, l'application de maintenance concerné par cette étude s'exécute par l'intermédiaire d'une JVM. Or, d'après [TLS-DRG 2002] les JVM ne sont pas exemptes de vulnérabilités. Ces vulnérabilités pourraient conduire à des fautes de mode commun sur notre architecture.

De ce fait, une simple comparaison basée sur des traces d'exécution ne permettra pas de détecter des erreurs de comportement. Cependant, il n'est pas nécessaire de pousser l'analyse jusqu'à la valeur des variables et objet. Effectivement, différentes JVM peuvent amener des optimisations hétéroclites sur le bytecode, qui peuvent entrainer des évolutions distinctes des valeurs des variables et objets. La comparaison de toutes les variables et objets serait trop difficile. Par contre, le comportement de l'application vis-à-vis de son environnement d'exécution ne doit pas être modifié par ces différentes optimisations. Ce sont ces interactions (avec le clavier, le réseau, etc) qui doivent être comparées. Il s'agit donc d'intercepter les interactions avec l'environnement et de les confronter à une référence.

Nous mettons donc en pratique les principes de la détection basés sur une observation et une comparaison. Dans la suite de cette section, nous détaillons le modèle, l'instrumentation de l'application pour mener les observations et son utilisation pour analyser les observations.

III.2 Principe de la détection à l'exécution

Le principe de la détection à l'exécution repose sur la notion d'observateurs et de travailleurs tel que définie par [Ayache *et al.* 1979]. L'objectif est de vérifier que les spécifications de l'application exécutée ne sont pas violées suite à des fautes matérielles ou logicielles ou des attaques qui apparaitraient lors de l'exécution. Ces spécifications définissent le modèle d'exécution de l'application. Un modèle très simple peut, par exemple, se limiter à une liste d'enchainements de procédures. Le modèle d'exécution peut alors être considéré comme un séquencement d'actions élémentaires. Dans [Ayache *et al.* 1979], l'observateur est le programme qui compare les observations au modèle d'exécution. Le travailleur est le programme exécuté. Les étapes principales se résument à l'enchainement suivant :

- l'observation du comportement des différentes instances de l'application ;
- la comparaison du comportement des différentes instances de l'application vis-à-vis du modèle d'exécution de l'application.

Ainsi, tout comportement qui diffère du modèle d'exécution est signalé comme comportement anormal et les exécutions des différentes instances de l'application sont arrêtées.

Notre démarche s'appuie sur ce principe en utilisant la modélisation sous forme de graphe des différentes exécutions possibles de l'application. La comparaison des séquences d'actions élémentaires avec l'ensemble des séquences admises par le modèle est ensuite réalisée. Ainsi, une attaque aboutissant à une séquence d'actions élémentaires qui ne respecte pas le modèle sera détectée. En particulier, ce mécanisme peut détecter des attaques qui affectent les deux instances de l'application, lorsque l'attaque exploite une vulnérabilité commune aux systèmes d'exploitations.

III.3 Modèle d'exécution

III.3.1 Description des interactions observées

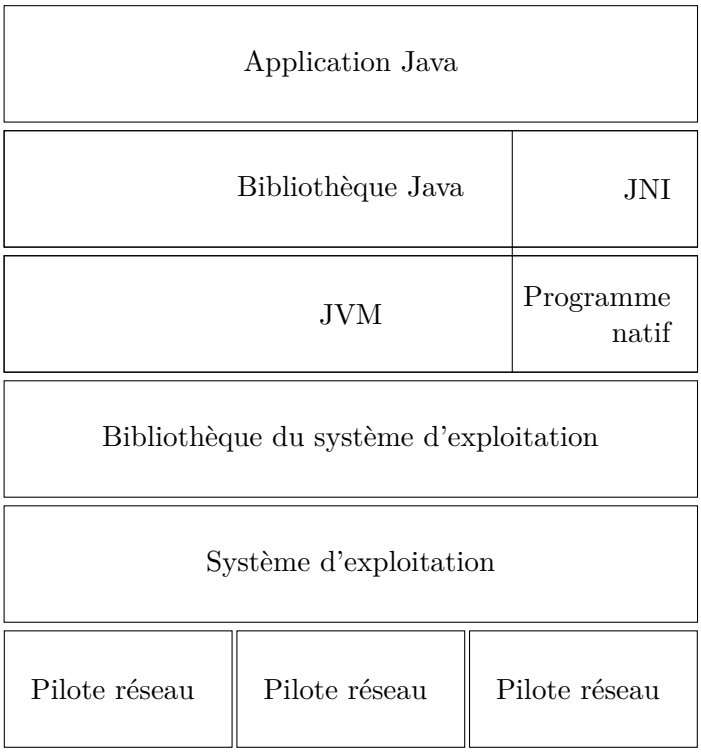

Figure III.1 – Architecture logicielle de la machine virtuelle

L'application considérée lors de cette étude est développée avec le langage Java. Elle est exécutée par une machine virtuelle Java. Ce choix garantit une bonne portabilité. De cette façon, le système d'exploitation utilisé doit seulement disposer de l'implémentation d'une JVM (spécifique au système d'exploitation) pour pouvoir être intégré dans l'architecture. Dans ce contexte, nous considérons que l'application Java interagit avec son environnement, composé essentiellement d'applications accessibles par le réseau, du système de fichier local à l'application et d'interfaces homme-machine (écran, clavier, souris). Ces accès sont tous réalisés par l'intermédiaire de la bibliothèque Java qui accompagne la JVM. La Figure III.1 présente l'architecture logicielle considérée sur une machine virtuelle.

Il existe principalement deux manières d'invoquer, depuis une application Java, les services mis à disposition par les composants logicielles sous-jacents : les invocations JNI et les invocations de la bibliothèque standard. Dans la suite, nous présentons ces deux types d'accès afin d'élaborer une stratégie d'interception des invocations.

Invocations JNI

Les invocations JNI [Sheng 1999] permettent d'enrichir les services mis à disposition des applications Java avec de nouveaux services développés en langage natif. Dans ce cadre, le concepteur développe deux composants : le service dans un langage natif, en général, en langage C et développe un composant Java permettant aux applications Java d'invoquer ce service. Cette approche consiste donc à déporter hors de la JVM les traitements qui auraient pu être réalisés au sein de l'application Java. L'avantage de cette approche est que le service développé peut profiter directement des services du système d'exploitation sans être contraint par les vérifications réalisées par la JVM. L'inconvénient réside dans la diminution de la portabilité de l'application Java. L'application ne peut alors être exécutée que sur un environnement où les invocations JNI sont possibles et sur lequel le service natif aura été porté.

Invocations de la bibliothèque standard Java

Une JVM est accompagnée d'une bibliothèque Java standard offrant aux applications les services usuels : accès au système de fichiers, accès au réseau, accès aux entrées/sorties, etc. L'avantage de cette approche est le maintien de la portabilité de l'application. Une application développée et testée sur une JVM fonctionnera sur toutes les autres JVMs disposant de la bibliothèque standard. L'inconvénient est que les applications ne peuvent se baser que sur les services offerts par cette bibliothèque.

Stratégie d'interception des invocations

L'architecture présentée dans le Chapitre II Section II.4 est basée sur la diversification des environnements d'exécution. De plus, elle permet de comparer les interactions entre les instances de l'application Java et leur environnement. Aussi, l'utilisation de l'approche JNI entraînerait l'exécution d'un programme en langage natif que nous ne serions pas en mesure d'observer. Par conséquent, notre architecture ne permet de prendre en compte que des applications utilisant uniquement la bibliothèque Java standard. Mais n'ayant aucune maîtrise sur l'application Java à exécuter, nous devons quand même envisager qu'elle puisse contenir des invocations JNI. Dans ce cas, afin de protéger l'architecture, les seules interactions autorisées entre l'environnement d'exécution et l'extérieur doivent être des invocations liées à la bibliothèque standard de Java. Les autres invocations d'entrée-sortie (invocations JNI) sont bloquées par l'hyperviseur. En revanche, l'application peut faire des invocations JNI qui ne soient pas liées aux entrées-sorties, à condition que les programmes natifs correspondants aient été portés sur les deux systèmes d'exploitation et que les deux instances correspondantes aient un comportement identique.

Les interactions autorisées et les interactions bloquées sont illustrées sur la Figure III.2.

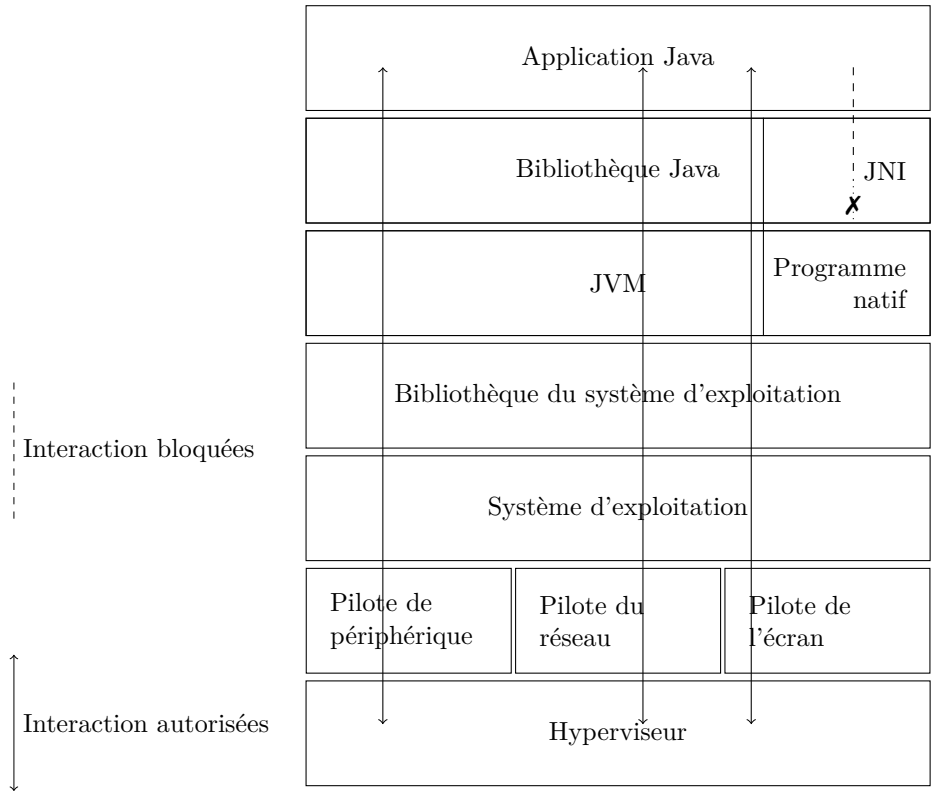

Figure III.2 – Interactions autorisées et bloquées par l'hyperviseur

Constructeurs et méthodes capturés

Dans cette section nous rappelons la stratégie d'interception retenue présenté au Chapitre II Section II.4.2. Cette stratégie est décrite par les six points suivants :

1. Injection de code dans le code du constructeur d'une classe qui hérite directement d'une classe d'entrée/sortie.
2. Injection de code autour de l'invocation des méthodes non surchargées d'une classe d'entrée/sortie.
3. Injection de code autour de l'invocation de l'instanciation d'une classe d'entrée/sortie.
4. Injection de code au début des méthodes susceptibles d'être invoquées directement par la bibliothèque Java (méthodes nommées *callback*, correspondant, par exemple, à la méthode invoquée lors de l'action sur un bouton).
5. Injection de code dans le constructeur d'une classe qui n'hérite pas d'une classe d'entrée/sortie, qui n'implémente pas indirectement une interface d'entrée/sortie mais qui implémente directement une interface d'entrée/sortie.
6. Injection autour de la méthode run d'une classe de type *Thread* ou *Runnable*.

En respectant scrupuleusement cette stratégie, il devient inutile de traiter les cas suivants :

A Injection de code dans le constructeur d'une classe héritant indirectement d'une classe d'entrée/sortie : ce cas sera pris en compte par le cas 1 appliqué à la classe parente héritant directement de cette classe d'entrée/sortie.

B Injection de code autour de l'invocation d'une méthode d'entrée/sortie surchargée : ce cas sera traité par le cas 2.

Cette stratégie est illustrée dans la Figure III.3. Sur cette figure, un trait avec une double flèche correspond à une méthode qui est interceptée. Un trait avec une flèche pleine correspond à une relation d'héritage et un trait en pointillés avec une flèche correspond à une invocation. Ce formalisme est proche du formalisme du langage UML.

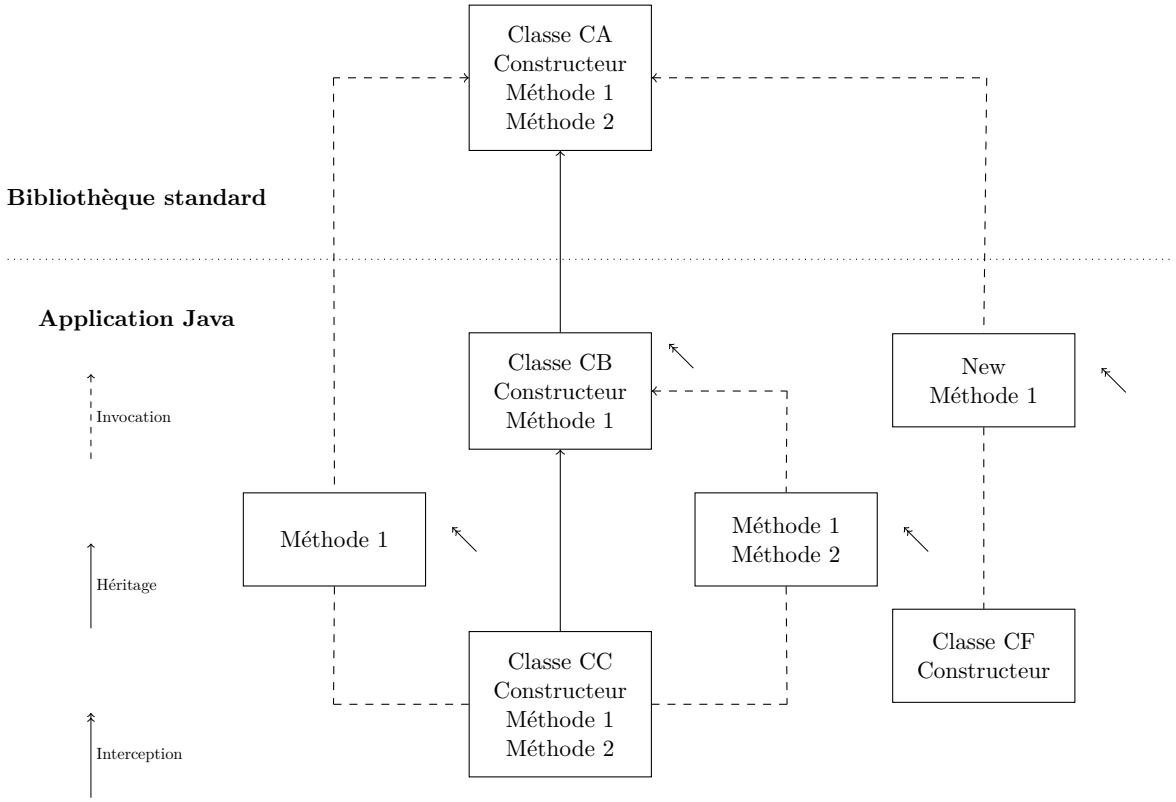

Figure III.3 – Stratégie d'interception des méthodes et constructeurs

III.3.2 Construction du modèle d'exécution

Le modèle d'exécution doit permettre de représenter l'ensemble des traces que les exécutions cohérentes du programme peuvent générer. Avant de détailler ce modèle, nous devons établir des hypothèses quant à son obtention. En effet, le modèle ne sera pas le même s'il est fourni par un tiers ou déduit de spécifications ou bien du bytecode de l'application. Nous considérons les possibilités suivantes pour chacun de ces cas :

Possibilité 1 : Modèle fourni par le bureau d'études
Dans ce cas, le modèle est pris en compte de manière transparente, sans nécessiter de modification de notre part. Il présente un ensemble des comportements attendus par le développeur, indépendamment de l'implémentation de l'application. Par contre, elle présente l'inconvénient d'alourdir la charge de travail du bureau d'études. Or, nous souhaitons conserver la transparence de l'architecture à ce niveau. Autrement dit, le bureau d'études en charge du développement de l'application de maintenance ne doit pas avoir à se soucier de la présence de l'architecture de sécurité.

Possibilité 2 : Modèle déduit des spécifications formelles de l'application
Cette possibilité présente le même avantage que la première. Son inconvénient se situe au niveau de la disponibilité de la spécification. De plus, le langage employé pour exprimer cette spécification doit être le même pour toutes les applications, ceci peut être difficile à réaliser si plusieurs équipes sont en charge du développement. Notre architecture doit pouvoir fonctionner quelle que soit l'application à exécuter, que cette dernière soit livrée avec les spécifications ou non.

Possibilité 3 : Modèle déduit du bytecode de l'application
L'ensemble des comportements obtenu par l'analyse du bytecode permet à notre architecture de fonctionner quelle que soit l'application. De plus, cet ensemble de comportement représente ce que fait l'application et non ce que le développeur à voulu lui faire faire.

Afin d'obtenir le modèle d'exécution, nous privilégions la troisième possibilité et étudions donc le fichier binaire de l'application une fois celui-ci téléchargé sur le portable de maintenance.

Ce modèle est représenté sous la forme d'un graphe de contrôle. Un graphe de contrôle est un modèle employé dans la théorie de la compilation [Aho *et al.* 2006] et de la décompilation [Cifuentes & Gough 1995]. Dans notre utilisation, un nœud représente un état d'attente pour l'application, un arc entre deux nœuds représente une invocation à une opération de la bibliothèque graphique. Un nœud peut posséder plusieurs arcs en sortie, notamment pour représenter les structures conditionnelles du type *if/then/else, while*, etc. Inversement, un nœud peut posséder plusieurs arcs en entrée, lorsque différentes branches de structures conditionnelles convergent vers ce nœud.

L'élaboration du modèle se base sur l'analyse du binaire de l'application. Toutefois, afin de faciliter la compréhension du principe, nous présentons notre démarche en s'appuyant sur le code source de l'application.

Les Figures III.4 et III.5 représentent un exemple d'une méthode Java et le graphe de contrôle associé. Ce code débute par la déclaration de la fonction Java qui se traduit par l'état q_0 sur le graphe de contrôle. La fonction se poursuit par la création de deux boutons, ce qui se traduit par les états q_1 et q_2. Après la création du second bouton, l'exécution peut se poursuivre soit par l'invocation de l'opération $a.setVisible(m)$ soit par l'invocation de $b.setSize(2, 5)$, selon le résultat du test booléen utilisé par la structure *if*. Par conséquent, deux arcs sortant de q_2 sont créés pour pointer vers q_3a et q_3b. Pour finir, les deux branches de la structure de contrôle *if* convergent vers l'état final q_6, qui marque la fin du graphe de contrôle associé à cette méthode.

Il existe autant de traces qui empruntent la branche *if* de la structure de contrôle, qu'il y a de valeurs différentes pour la variable m. Au contraire, il n'existe qu'une seule trace correspondant à l'exécution de la branche *else* de la structure de contrôle.

```
void init(int n, String m)
{
  a = new Button();
  b = new Button();
  if (n == 0) {
    a.setLabel(m);
    a.setVisible(true);
  } else {
    b.setSize(2, 5);
    b.setVisible(true);
    b.setName("i");
  }
}
```

```
0:  new       #2; //class java/awt/Button
3:  dup
4:  invokespecial   #3; //Method Button."<init>"
7:  astore_2
8:  new       #2; //class Button
11: dup
12: invokespecial   #3; //Method Button."<init>"
15: astore_3
16: iload_1
17: ifne      33
20: aload_2
21: aload_0
22: invokevirtual   #4; //Method Button.setLabel
25: aload_2
26: iconst_1
27: invokevirtual   #5; //Method Button.setVisible
30: goto      50
33: aload_3
34: iconst_2
35: iconst_5
36: invokevirtual   #6; //Method Button.setSize
39: aload_3
40: iconst_1
41: invokevirtual   #5; //Method Button.setVisible
44: aload_3
45: ldc       #7; //String i
47: invokevirtual   #8; //Method Button.setName
50: return
```

Figure III.4 – Code source & Bytecode Java

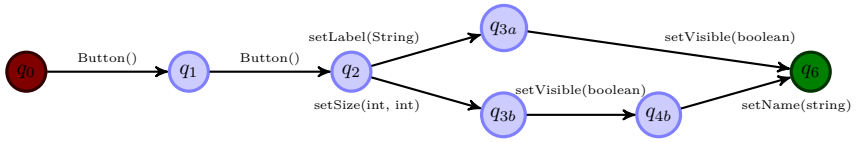

Figure III.5 – Graphe de contrôle

III.3.3 Utilisation du modèle

Comme nous venons de le décrire, notre modèle est crée à partir du bytecode Java de l'application. Le modèle est utilisé pour comparer chaque exécution de l'application sur les machines virtuelles. Nous prenons en exemple le graphe de contrôle de la Figure III.5 afin de préciser son utilisation. La Figure III.6 représente une trace d'exécution de l'application sur une machine virtuelle. En comparant la trace d'exécution aux noms des invocations portées par les arcs, nous pouvons déterminer le chemin du modèle correspondant à cette trace (cf. Figure III.7). Étant donné qu'il existe un chemin identique dans le modèle, nous concluons que la trace d'exécution est valide.

Button() Button() SetLabel(String) SetVisible(boolean)

Figure III.6 – Trace d'exécution 1

La trace d'exécution déduite de la Figure III.6 débute par deux invocations successives de la méthode `Button`, suivi des invocations des méthodes `setLabel(m)` et `setVisible(true)`.

A partir de cette trace, nous pouvons construire le chemin proposé sur la Figure III.7. Celui-ci débute par l'état q_0. L'appel de la méthode `Button` permet d'activer l'état q_1 et q_2. Ensuite, la méthode `setLabel(string)` est nécessaire pour atteindre l'état q_{3a}. La dernière méthode `setVisible(boolean)` assure la transition jusqu'à l'état q_6. Cette séquence d'états (q_0, q_1, q_2, q_{3a} *et* q_6) ainsi que les méthodes associées nous permettent de valider cette exécution en accord avec le graphe de contrôle présenté par la Figure III.5.

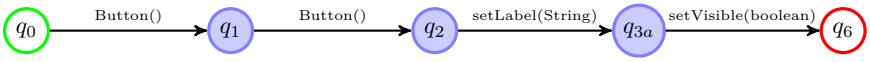

Figure III.7 – Exemple d'une exécution valide

Prenons l'exemple de la trace d'exécution présentée dans la Figure III.8, elle débute par deux invocations successives de la méthode `Button`, suivi de l'invocation de la méthode `setVisible(true)`. A partir de cette trace, il nous est impossible de construire un chemin admis par le graphe de contrôle de la Figure III.5. Nous en concluons que cette trace d'exécution est non valide.

Button() Button() SetVisible(boolean)

Figure III.8 – Trace d'exécution 2

III.4 Instrumentation automatique de l'application

Comme décrit dans le Chapitre II Section II.4.2, nous avons fait le choix d'instrumenter le binaire de l'application au lieu de modifier la JVM par souci d'efficacité et de portabilité. Dans cette section, nous présentons les différentes possibilités afin de réaliser l'instrumentation de l'application et ainsi créer le modèle d'exécution de référence. Il s'agit de la programmation orientée aspect et de l'injection de code.

Utilisation de la programmation orienté aspect

La programmation orientée aspect, par l'intermédiaire d'outil comme aspectJ [Kiczales *et al.* 2001], permet l'instrumentation au niveau du code source et du binaire d'une application. Un aspect est une entité logicielle qui capture une fonctionnalité d'une application. Il est défini par trois éléments principaux :

- Un point de jonction (*joinpoint*) représente un point dans le flot de contrôle de l'application ;
- Une coupe (*pointcut*) sélectionne un ensemble de points de jonction afin de définir l'endroit du code de l'application où doit être intégré l'aspect ;
- Un code (*advice code*) correspond à un bloc de code définissant le comportement d'un aspect.

Le tissage (*weaving*) est un processus qui prend en entrée un ensemble d'aspects et une application et fournit en sortie une application dont le comportement et la structure sont étendus par les aspects.

Cependant, nous n'employons pas cette approche car d'une part, il est nécessaire de déployer sur la machine virtuelle les bibliothèques correspondantes à AspectJ. D'autre part, certaines propriétés des classes peuvent ne pas être facilement exprimables (il est par exemple difficile d'exprimer les cas numéro 2 présenté au paragraphe III.3.1).

Nous avons donc opté pour une analyse statique du bytecode afin de permettre une injection de code.

Injection de code

La technique d'injection de code est mise en œuvre en utilisant la bibliothèque ASM [Bruneton *et al.* 2002]. Cette bibliothèque permet de lire le bytecode de différentes classe et d'y insérer du nouveau bytecode. Le résultat peut être alors déployé sur la machine virtuelle sans nécessiter le déploiement de bibliothèques supplémentaires.

Rappelons que le langage Java est un langage à pile, les paramètres à fournir à la méthode invoquée doivent être au préalable empilés dans la pile. Ensuite, ces paramètres sont récupérés auprès d'une structure complexe de la JVM appelée le *Constant Pool* (CP). L'interception de la méthode est réalisée en injectant une invocation à l'hyperviseur, en fournissant en argument une copie des arguments de la méthode observée. Il est important de restaurer intégralement l'état de la pile avant l'invocation de la méthode observée. Nous pouvons adopter deux approches différentes : L'injection de paramètres ou la duplication de paramètres.

La première approche consiste à injecter, avant l'empilement des paramètres de la méthode observée, le bytecode permettant de récupérer les paramètres depuis le CP et d'invoquer l'hyperviseur. Cette approche présente l'inconvénient de devoir analyser le contenu du CP pour toutes les méthodes observées.

La seconde approche consiste à profiter de l'empilement des paramètres de l'invocation de la méthode observée, pour dupliquer ces paramètres et invoquer l'hyperviseur. Autrement dit, le sommet de la pile est dupliqué.

Cette seconde approche a été retenue et est illustrée par la Figure III.9. Après le traitement de la classe, le code correspondant à l'injection est placé entre l'empilement des paramètres et l'invocation de la méthode. Cette approche modifie le contenu de la pile. Ces modifications sont représentées sur la Figure III.11.

Dans l'exemple A (Figure III.10), l'évolution de la pile est représenté pour l'exécution de la fonction `SetTitle()` sans interception. L'objet de l'invocation et les paramètres de l'invocation sont stockés dans la pile (ligne 24 et 25). Lors de l'invocation, ligne 27, la signature de la méthode invoquée permet à la JVM de connaitre les différents éléments au sommet de la pile nécessaire pour réaliser l'invocation. Après cette invocation, ces éléments sont dépilés.

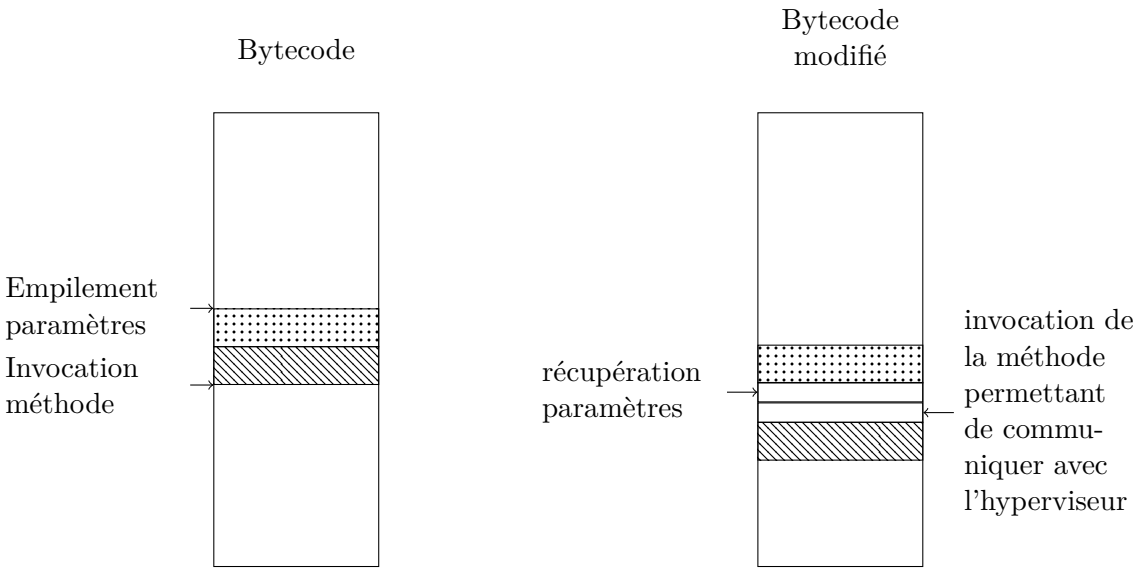

Figure III.9 – Instrumentation du bytecode - duplication des paramètres

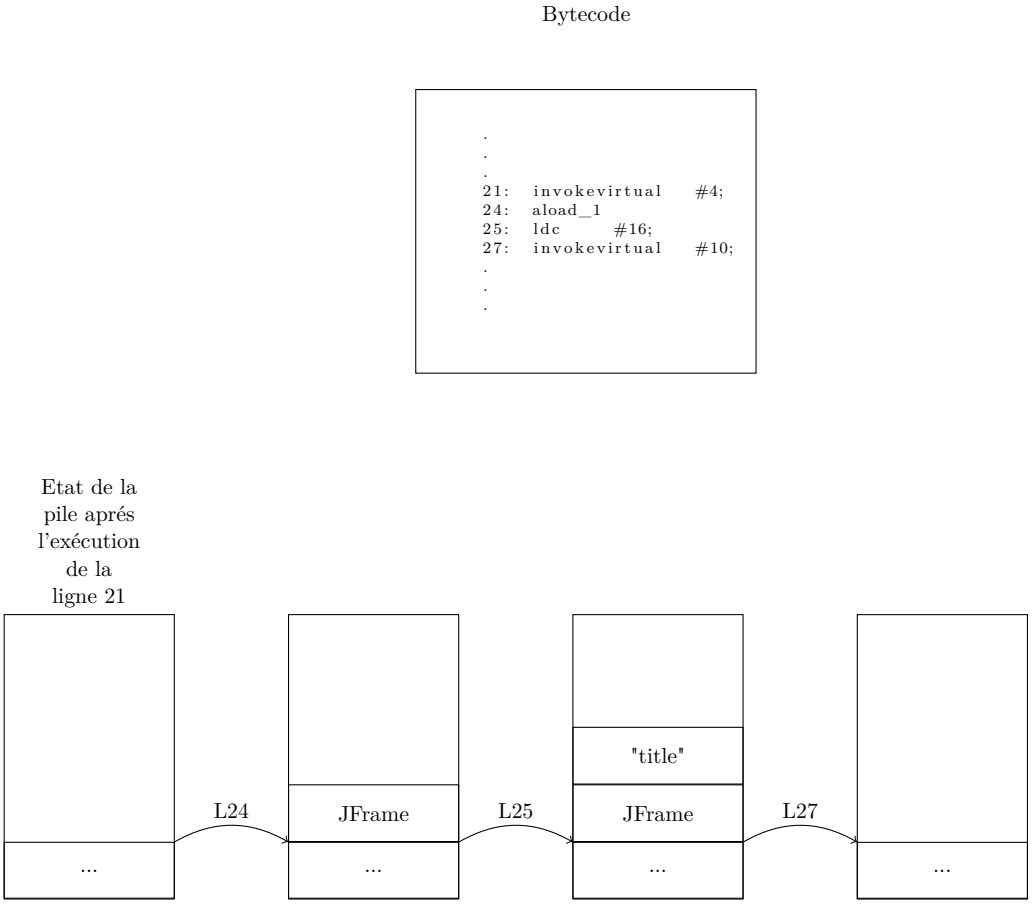

Figure III.10 – Exemple A - évolution de la pile pour exécution sans interception.

Dans l'exemple B (Figure III.11), l'évolution de la pile est représentée pour l'exécution de la fonction `SetTitle()` avec interception. L'objet de l'invocation et les paramètres de

l'invocation sont stockés dans la pile (ligne 24 et 25).

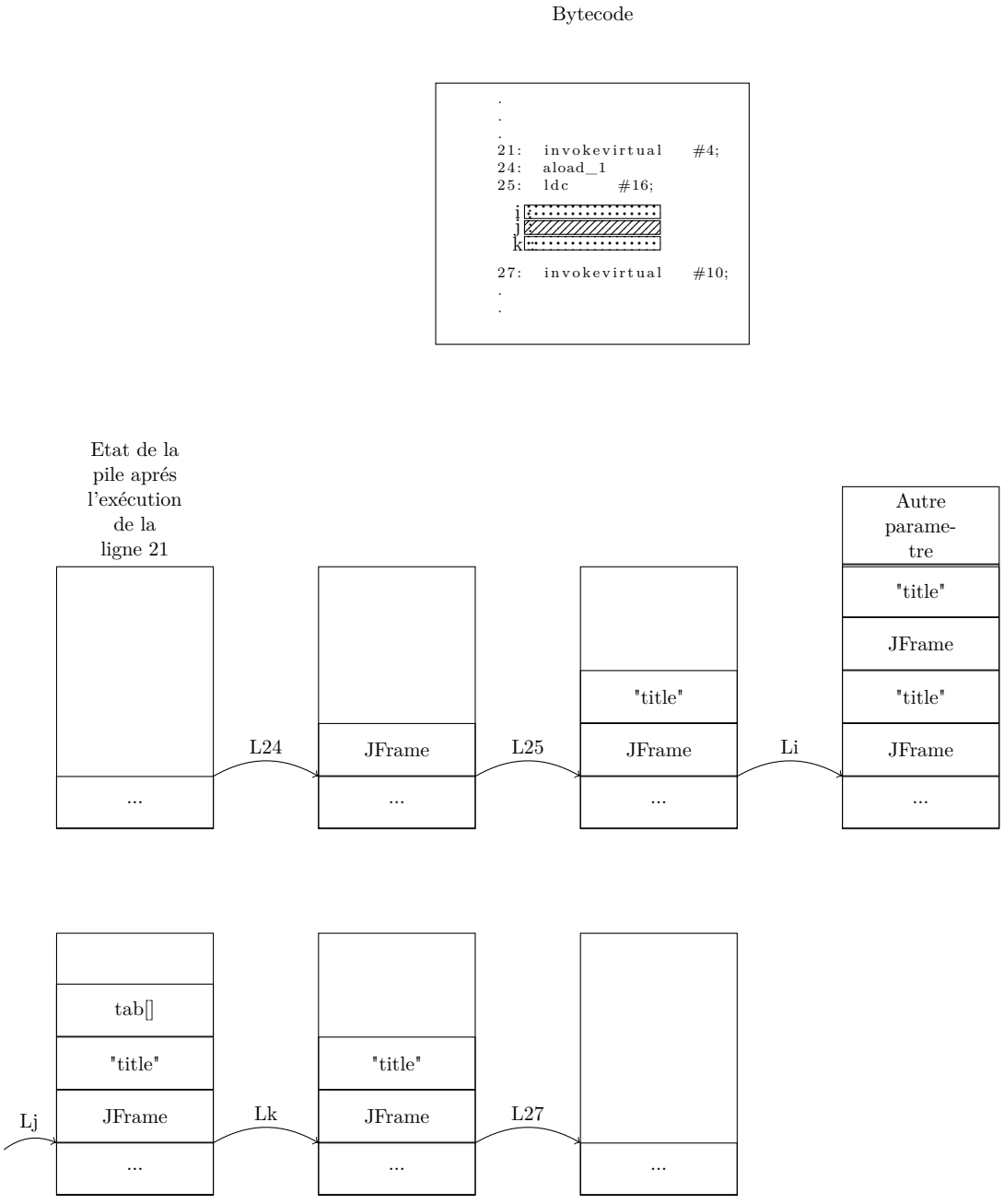

Figure III.11 – Exemple B - évolution de la pile pour exécution avec interception.

III.5 Complémentarité des méthodes de comparaison

Nous avons présenté au Chapitre II Section II.4.2 une première méthode de comparaison. Celle-ci est basée sur l'analyse des traces d'exécution entre différentes instances d'une même application. Les traces d'exécution de chaque instances sont comparées entre elles. La deuxième méthode, présentée à la Section III.2, se base sur un modèle d'exécution de l'application. Les traces d'exécution issues des différentes instances sont comparées une à une avec le modèle.

Nous rappelons que les traces d'exécution de l'application concernent les appels de méthodes graphiques. Le Tableau III.1 présente un exemple de trace d'exécution composée de quatre messages.

$Trace_A$	Method name	Parameter
new Button()	Button	null
new Button()	Button	null
setValue(12)	setValue	12
setVisible(True)	setVisible	True

Tableau III.1 – Exemple de trace d'exécution

Le Tableau III.2 présente différentes traces d'exécution qui peuvent être observées. Les traces $Trace_1$ et $Trace_3$ correspondent à différentes exécutions qui activent la branche *if*. Elles différent au niveau de la valeur de m. Les traces $Trace_2$ et $Trace_4$ correspondent à aucune exécution correcte de l'application.

$Trace_1$	$Trace_2$
new Bouton()	new Bouton()
new Bouton()	new Bouton()
setValue(12)	setVisible(True)
setVisible(True)	setValue(12)
$Trace_3$	$Trace_4$
new Bouton()	new Bouton()
new Bouton()	setName(i)
setValue(13)	setSize(2,5)
setVisible(True)	setVisible(True)
	new Bouton()

Tableau III.2 – Traces d'exécution

Ces traces correspondent à l'observation de l'exécution de l'application sur une des machines virtuelles. Lors de l'exécution sur l'architecture diversifiée, nous obtenons ainsi deux traces à comparer, chacune pouvant être soit $Trace_1$, soit $Trace_2$, soit $Trace_3$ soit $Trace_4$. Le Tableau III.3 présente les différents cas de figure liés à l'observation de ces traces sur l'architecture diversifiée.

Nous rappelons que nous utilisons deux mécanismes de comparaison afin de décider de la validité d'une trace. Tout d'abord les traces sont comparées entre elles, puis chacune

d'elle est comparée vis-à-vis du modèle d'exécution. Le Tableau III.3 présente tous les cas possibles sous la forme d'une table de vérité.
Les deux premières colonnes correspondent aux traces observées sur les machines virtuelles. La troisième colonne correspond au diagnostic de la première méthode de comparaison. Les deux colonnes suivantes correspondent au diagnostic lié à l'utilisation du modèle d'exécution (l'abréviation *ME* signifie Modèle d'Exécution), soit la deuxième méthode de comparaison. Enfin, la dernière colonne correspond au verdict prit sur les traces d'exécution. Ce verdict est prononcé en fonction d'un `OU logique` du diagnostic des deux mécanismes mis en jeu. Le verdict répond à la question : le système a t-il subit une tentative d'attaque ? La réponse peut être oui ou non.

	VM_1	VM_2	$VM_1 - VM_2$	$VM_1 -$ ME	$VM_2 \times$ ME	Verdict
1	$Trace_1$	$Trace_1$	1	1	1	non
2			1	1	0	
3			1	0	1	
4	$Trace_2$	$Trace_2$	1	0	0	oui
5	$Trace_1$	$Trace_3$	0	1	1	oui
6	$Trace_1$	$Trace_2$	0	1	0	oui
7	$Trace_2$	$Trace_1$	0	0	1	oui
8	$Trace_2$	$Trace_4$	0	0	0	oui

1 Comparaison réussie 0 Comparaison échouée

Tableau III.3 – Confrontation des mécanismes de comparaison

Les comparaisons échouées désignées par 0 signifient que pour la méthode considérée, les traces d'exécution ne sont pas équivalentes (pour la première méthode) ou bien qu'elles ne sont pas en accord avec le modèle (pour la seconde méthode). En d'autre terme, une tentative d'attaque s'est produite et a été détectée.

Les comparaisons réussies désignées par 1 signifient que pour la méthode considérée, les traces d'exécution sont équivalentes (pour la première méthode) ou bien qu'elles sont en accord avec le modèle (pour la seconde méthode). En d'autre terme, aucune tentative d'attaque ne s'est produite.

Les traces générées par les machines virtuelles, dans la ligne 1 sont identiques et correspondent à des traces cohérentes vis-à-vis du modèle. Dans ce cas, les deux méthodes diagnostiquent aucune tentative d'attaque. Les lignes 2 et 3 sont à notre connaissance impossible à réaliser dans le contexte de notre étude. En effet, si les deux traces sont déclarées identiques par le premier mécanisme, le mécanisme de détection basé sur le modèle ne devrait pas émettre deux diagnostic contraire. Les autres cas correspondent à l'occurrence d'une attaque : soit les deux traces sont différentes soit elles ne sont pas cohérentes vis à vis du modèle. La comparaison des traces entre elles se base sur l'ordre et les valeurs des observations. Elle détecte toutes les attaques sauf celles correspondant au cas de la ligne 4 ($Trace_2$; $Trace_2$). Quant à la comparaison des traces vis-à-vis du modèle d'exécution, elle permet d'identifier le cas de la ligne 4 ($Trace_2$; $Trace_2$) comme étant une attaque. Par contre, elle manque la détection du cas de la ligne 5 ($Trace_1$; $Trace_3$) car une comparaison des valeurs est nécessaire. Les deux méthodes sont donc complémentaires et permettent de détecter tous les cas d'attaque cités. Les lignes 6 et 7 sont équivalentes puisque seule une

inversion des traces est effectuée. La ligne 8 correspond à l'éventualité ou les deux traces d'exécutions seraient à la fois différentes entre elles et vis-à-vis du modèle d'exécution.

Conclusion

Ce chapitre présente notre proposition concernant une méthode de détection basé sur un modèle d'exécution d'une application. Nous souhaitons par l'ajout de cette méthode, améliorer le mécanisme de détection présenté au Chapitre II Section II.4.2 en augmentant la couverture de détection.

Nous avons débuté ce chapitre par la présentation du principe de détection à l'exécution afin de décrire la méthode employée lors de notre étude. Puis, nous avons détaillé cette méthode en débutant par la description des événements à observer et nous avons proposé une stratégie d'interception de ces évènements.
La construction du modèle d'exécution décrite par la suite a présentée les différentes possibilités. Nous avons opté pour une construction du modèle déduite de l'analyse du bytecode de l'application, cela nous a permis d'être indépendant de l'application que nous avions à prendre en compte.
Nous avons illustré, ensuite, la méthode d'utilisation du modèle d'exécution ainsi créé par différents exemples. Cette illustration a validé, au minimum de manière théorique notre approche et a montré sa simplicité d'utilisation.
Nous avons poursuivi ce chapitre par une description de l'instrumentation de l'application. Ce point, plus technique, a présenté notre choix stratégique afin d'optimiser notre méthode.
Un dernier exemple nous a permit de discuter pour conclure à la complémentarité des méthodes employées.

Chapitre IV

Virtualisation : définition et expérimentation

Introduction

Dés le début des années 2000, la virtualisation a connu un essor important et s'est imposée sur le marché des logiciels comme un ensemble de techniques matérielles ou logicielles permettant de partager les ressources d'un ordinateur pour de multiples environnements d'exécution.

La virtualisation est une technique qui prend ses origines dans les laboratoires d'IBM dans les années 1960 avec le système d'exploitation cp-40 utilisé par le *mainframe* S/360. Le système cp-40 était initialement prévu pour partitionner un grand *mainframe* en plusieurs partitions logiques tout en utilisant une seule machine physique. L'introduction du concept d'Architecture de Jeux d'Instruction, ISA (*Instruction Set Architecture*) a initié la séparation entre partie logicielle et matérielle. Ainsi, ces deux parties peuvent évoluer indépendamment l'une de l'autre, seule la couche logicielle assurant la traduction entre la partie logicielle et la partie matérielle doit être modifiée en conséquence.

La virtualisation peut être définie comme une technique permettant de séparer les ressources d'une machine afin de fournir à de multiples environnements d'exécution la possibilité de cohabiter sur une même machine matérielle.

Nous débutons ce chapitre par la présentation de différents types de virtualisation avant de détailler les moyens techniques d'aide à la virtualisation processeur. Nous décrivons dans un deuxième temps le banc de tests que nous avons utilisé afin d'évaluer une solution de virtualisation.

IV.1 Types et techniques de virtualisation

IV.1.1 Types de virtualisation

Il existe deux principaux types de virtualisation, la virtualisation de processus et la virtualisation de système [Smith & Nair 2005].

La **virtualisation de processus** consiste pour un système d'exploitation, à virtualiser un espace mémoire, un processeur ainsi que les autres ressources du système pour chaque processus exécuté. Chacun des processus interagit avec le système d'exploitation par l'intermédiaire d'interfaces binaires d'application, *Application Binary Interface* (ABI), ou d'interfaces de programmation, *Application Programming Interface* (API) en ignorant les activités des autres processus.

La **virtualisation de système** s'applique sur un système entier par l'utilisation d'une couche de virtualisation qui permet à de multiples systèmes de s'exécuter isolément les uns des autres tout en étant sur la même machine physique. La couche de virtualisation porte le nom de moniteur de machine virtuelle ou d'hyperviseur.

La suite de cette section se concentre sur ce type de virtualisation.

Virtualisation d'exécution

Différentes implémentations de virtualisation de système ont été proposées. Nous présentons dans ce qui suit les principales implémentations : la virtualisation complète, la virtualisation native et la paravirtualisation.

La virtualisation complète (type 2) définit une approche dans laquelle l'hyperviseur s'exécute au dessus du système d'exploitation dit hôte, comme une simple application. Cela permet à la machine virtuelle, dans laquelle s'exécute un système d'exploitation dit invité et ses applications, d'utiliser le matériel virtuel fourni par l'hyperviseur. Dans cette configuration, des périphériques d'entrées/sorties sont attribués à la machine hôte en imitant les périphériques physiques dans l'hyperviseur. Les interactions avec ceux-ci dans l'environnement virtuel sont alors dirigées vers les dispositifs physiques réels. Cette architecture est représentée par la Figure IV.1.

Le principal avantage de cette approche réside dans sa facilité d'utilisation. En effet, un utilisateur peut installer une couche de virtualisation telle que VirtualBox ou VMware Workstation comme n'importe quel autre produit logiciel sur le système d'exploitation. La couche de virtualisation permet l'installation, dans un environnement virtualisé, d'un système d'exploitation invité ainsi que son utilisation comme s'il était directement exécuté sur le matériel.

Le principal inconvénient de cette architecture revient à la dégradation des performances par rapport à un système non virtualisé.

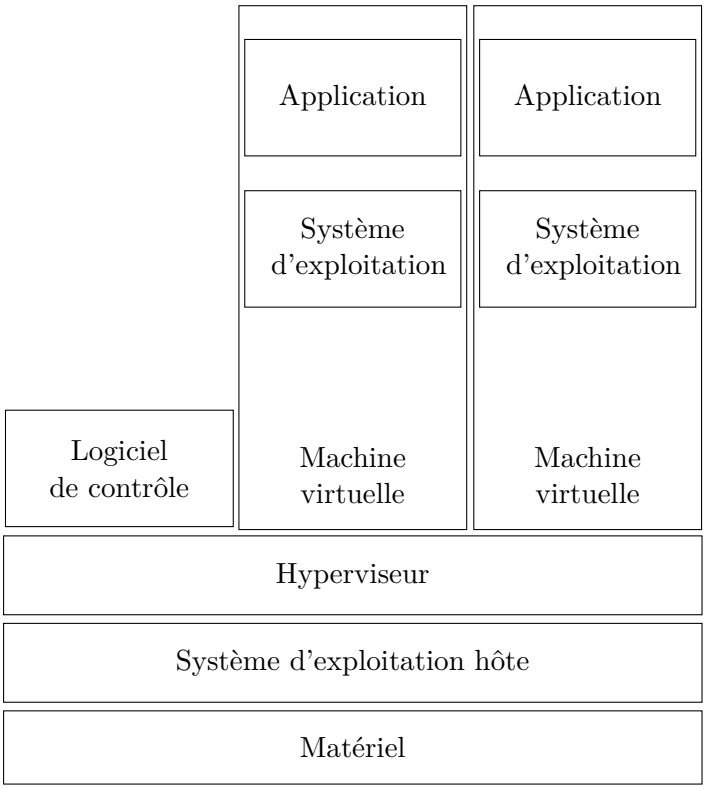

Figure IV.1 – Virtualisation complète

La virtualisation native (type 1) définit une approche dans laquelle l'hyperviseur s'exécute directement au-dessus du matériel tout en permettant au système d'exploitation invité d'utiliser par son intermédiaire les ressources matérielles de la machine. La Figure IV.2 représente cette architecture.

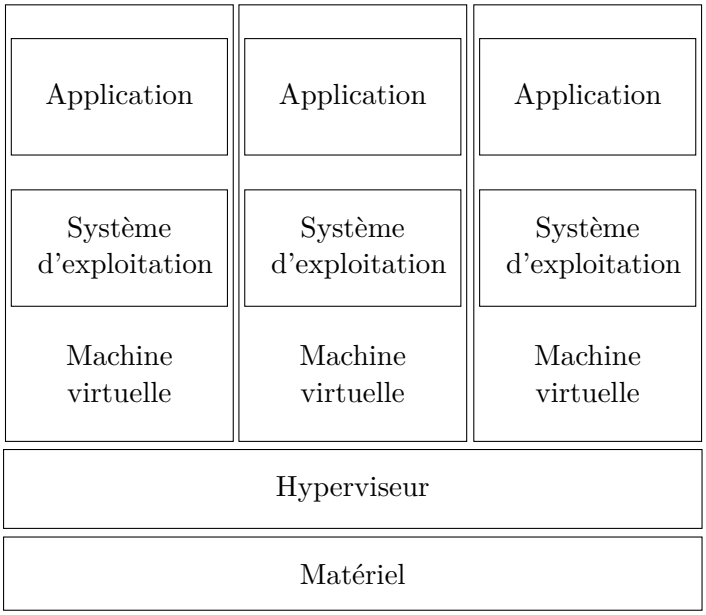

Figure IV.2 – Virtualisation native

La paravirtualisation utilise comme la virtualisation complète un système d'exploitation complet qui s'exécute sur le matériel émulé par une machine virtuelle, cette dernière s'exécutant au dessus d'un système hôte. Cependant, le système invité est modifié pour être exécuté par la machine virtuelle. Les modifications effectuées ont pour objectif d'améliorer les performances de l'architecture. Ainsi, il est permit au système invité par l'utilisation d'une interface spécifique d'accéder au matériel par l'intermédiaire du système hote au lieu d'accéder au matériel virtuel via les couches d'abstraction.

En pratique, un système paravirtualisé possède des pilotes de périphériques modifiés, qui lui permettent de communiquer directement avec le système hôte, sans avoir à passer par une couche d'abstraction. Les pilotes paravirtualisés échangent directement des données avec le système hôte, sans avoir à passer par une émulation du comportement du matériel. Certaines parties du système hôte doivent généralement être modifiées pour tirer profit de la paravirtualisation, en particulier la gestion de la mémoire et la gestion des E/S.

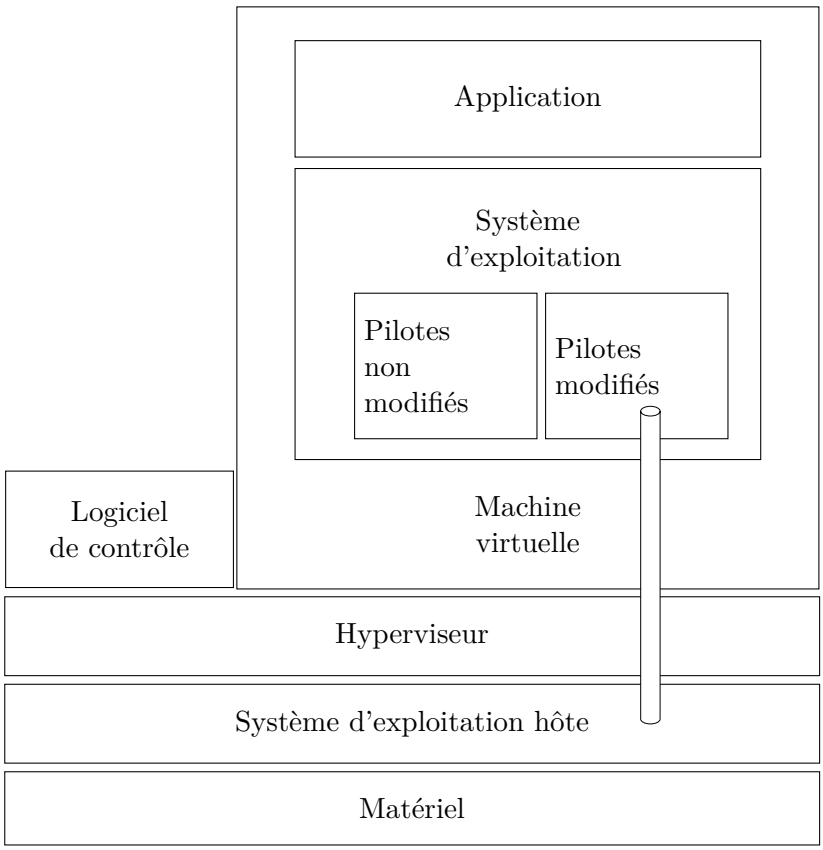

Figure IV.3 – Paravirtualisation

La Figure IV.3 décrit la structure d'une machine virtuelle et d'un système hôte supportant la paravirtualisation. Les pilotes non modifiés interagissent avec le matériel émulé par la machine virtuelle, tandis que les pilotes modifiés communiquent directement avec le système hôte. La simplification qui en résulte permet au système invité de collaborer plus efficacement avec l'hôte : les parties critiques du système communiquent presque directement avec le système hôte, en contournant les couches d'abstraction virtuelles (c'est-à-dire le matériel émulé).

Le reste de l'architecture est inchangé : la machine virtuelle est toujours une application utilisateur et le système d'exploitation hôte est toujours le seul à avoir un accès privilégié au matériel (cf. Figure IV.3).

IV.1.2 Moyens techniques d'aide à la virtualisation

Dans cette section, nous présentons les différentes technologies de virtualisation développées par les entreprises majeure du domaine : Intel et AMD. Intel Virtualization Technology (VT) est une famille de technologies supportant la virtualisation pour des plateformes Intel IA-32, Xeon et Itanium. Cela comprend des éléments pour le processeur, la mémoire et la virtualisation d'entrées/sorties. AMD-V offre des supports comparables pour le processeur, la mémoire et la virtualisation d'entrées/sorties. Pour ces différents composants nous détaillerons brièvement les technologies utilisées.

Virtualisation CPU

Le chipset 80286 d'Intel a été le premier de la gamme des architectures processeurs x86 à fournir deux principales méthodes d'adressage de la mémoire : un mode dit « réel » et un mode « protégé ». Un troisième mode dit « virtuel » permet plusieurs utilisations du mode « réel » en simultané. Le mode « réel » est rapidement devenu obsolète du fait de ses limitations : seulement 1Mo de mémoire peut être adressé et une seule application peut être exécutée à la fois. Le mode « virtuel » est également peu utilisé du fait de son adressage mémoire sur 16 bit et l'utilisation croissante de processeurs 32 voire 64 bits.

Le mode protégé comprend quatre niveaux de privilèges détaillés dans le Figure IV.4. Le niveau 0 (au centre de la figure) est le plus privilégié et est utilisé pour les segments de code les plus critiques d'un système, généralement le noyau d'un système d'exploitation. Les autres niveaux, avec progressivement moins de privilèges, sont utilisés par des segments de code moins critiques. Par exemple, les applications sont généralement exécutées au niveau 3. Bien que l'emploi de ces niveaux de privilèges soient conseillés par Intel, ils ne sont pas tous utilisés. En effet un système d'exploitation tel que Linux utilise seulement deux niveaux : le niveau 0 pour le noyau du système d'exploitation et le niveau 3 pour les applications.

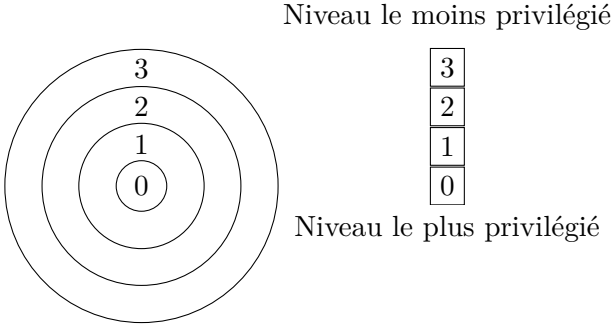

Figure IV.4 – Niveaux de privilèges du mode protégé de l'architecture Intel x86

Intel VT est connu sous le nom de VT-x pour les plateformes IA-32 et Xeon tandis qu'il prend le nom de VT-i pour la plateforme Itanium. La technologie VT-x ajoute deux nouveaux modes d'exécution au processeur, VMX *root* et VMX *non-root*. L'hyperviseur s'exécute en mode VMX *root* alors que les machines virtuelles s'exécutent en mode *VMX non-root*. Ces deux modes supportent les quatre niveaux de privilèges présentés dans le paragraphe précédent et permettent ainsi à l'hyperviseur une plus grande flexibilité pour le choix de privilèges à accorder à une machine virtuelle.

La transition entre ces deux modes est assurée par deux nouvelles instructions : VM *entry* qui assure la transition entre les modes VMX *root* et VMX *non–root*, et VM *exit* qui assure la transition entre les modes VMX *non-root* et VMX *root*. Ces instructions sont gérées par une nouvelle structure de données nommée *virtual-machine control structure* (VMCS).

AMD-v introduit également un mode d'exécution additionnel nommé « mode invité » qui est analogue au mode non-root d'Intel VT-x. L'accès au mode *invité* se fait par l'intermédiaire de l'instruction VMRUN. Lorsque l'instruction VMRUN est appelée par une machine virtuelle, l'accès au matériel est assuré par une structure de données nommée *Virtual Machine Control Block* (VMCB) correspondant à cette machine virtuelle. Cette structure enregistre les informations relatives aux événements et interruptions pouvant être interceptés par l'hyperviseur pour cette machine virtuelle.

Ces différentes technologies que sont Intel VT-x et AMD-v introduisent des extensions aux modes d'exécution existants.

Virtualisation mémoire

Les processeurs Intel x86 utilisent une table de pages afin de faire correspondre les adresses mémoires virtuelles aux adresses physiques, ces tables sont gérées par la *Memory Management Unit* (MMU) et sont principalement mises en cache dans le *Translation Lookkaside Buffe*r (TLB). Les OS invités utilisent les tables de pages des MMU émulées. Ces tables permettent au système invité d'avoir l'illusion d'assurer la correspondance entre les adresses virtuelles et les adresses physiques, or c'est l'hyperviseur qui réalise cette opération.

Un autre type d'adresse mémoire, nommé « pseudo-physique », est utilisé afin d'assurer le bon fonctionnement de la traduction. Le système d'exploitation invité assure la traduction entre les adresses virtuelles et les adresses « pseudo-physiques » tandis que l'hyperviseur maintient séparément des tables de traduction dites fantômes afin d'assurer la traduction des adresses pseudo-physiques en adresses physiques. Cependant, cette utilisation pose un problème de performance, car à chaque mise à jour des tables de page d'un système invité l'hyperviseur doit mettre à jour les pages fantômes.

Des mécanismes tel que les *Nested Page Tables* (NPT) d'AMD ou les *Extended Page Tables* (EPT) d'Intel permettent de palier ce problème de performance.

En effet, en utilisant un mécanisme de pagination imbriqué, le processeur enregistre à la fois l'adresse mémoire virtuelle et l'adresse physique de l'OS invité comme respective-

ment, adresse physique invitée et adresse physique réelle au sein du TLB. L'utilisation de l'identifieur d'adresse, *Address Space Identifier* (ASID), assure la correspondance entre les entrées du TLB et les machines virtuelles associées. De cette manière, des entrées de différentes machines virtuelles peuvent coexister au sein du TLB au même moment. Le processeur, par ce mécanisme, n'a plus qu'à synchroniser le TLB après chaque modification des pages de tables des OS invités.

Virtualisation d'entrées/sorties

La famille Intel VT comprend également un composant nommé Vt-d pour la virtualisation d'entrées/sorties. Ce composant permet par exemple d'allouer un périphérique à une seule machine virtuelle. L'accès au périphérique se fait sans contrôle de la part de l'hyperviseur.

IV.2 Évaluation des performances d'un hyperviseur

L'évaluation d'hyperviseur a été le sujet de nombreuses études. Certaines études comparent les hyperviseurs entre eux alors que d'autres s'intéressent de façon plus indirecte aux différences de performance entre une application s'exécutant sur un hyperviseur et cette même application s'exécutant sur un système d'exploitation classique.

Dans cette section, nous décrivons différentes études afin de présenter les méthodes d'évaluations utilisées et de mettre en évidence l'innovation apportée par notre méthode d'évaluation.

Les auteurs de [Xu *et al.* 2008] comparent la dégradation du temps d'exécution d'une application entre une machine physique exécutant un système d'exploitation non virtualisé et différentes solutions de virtualisation. Le logiciel Ubench [PhysTech 2005] est utilisé pour effectuer les mesures. La machine physique sert de référence afin d'évaluer les différences entre les hyperviseurs considérés.

Dans [McDougall & Anderson 2010], différentes techniques de virtualisation sont comparées en utilisant également comme référence un système d'exploitation non virtualisé. Dans cette étude, le logiciel VMark [Herndon *et al.* 2006] est utilisé pour évaluer les performances. Le logiciel NAS [Jin *et al.* 1999] est choisi dans [Bhukya *et al.* 2010] afin de comparer l'hyperviseur Xen et une solution issue de VMware [Walters 1999]. L'auteur examine en particulier l'influence du nombre de processeurs assignés à l'hyperviseur.

Dans [Umeno *et al.* 2006], les auteurs comparent les performances d'une configuration avec et sans utilisation de la virtualisation. Ils s'intéressent plus particulièrement aux performances processeurs, mémoires et réseaux. Pour ce faire ils utilisent l'algorithme de calcul BYTEMark [Seungkwon & Youngil 2007] pour mesurer le taux de charge des processeurs, le logiciel LMbench [McVoy & Staelin 1996] permet d'estimer le taux de transfert mémoire tandis que le logiciel Iozone mesure la bande passante réseau. Les résultats obtenus leur permettent d'évaluer la stabilisation du serveur en utilisant des machines virtuelles.

Le logiciel RandomWriter et l'algorithme Sort [TASF 2007] sont utilisés dans [Kontagora & Gonzalez-Velez 2010] afin d'évaluer un réseau de machines virtuelles. Ce réseau est composé de dix-sept machines physiques. Chaque machine exécute VMWare Workstation pour fournir un système d'exploitation virtualisé. Une machine physique est définie comme maître ; les seize autres sont utilisées dans différentes configurations.

La précision des mesures de performances dépend de la manière dont le temps est mesuré. Dans les études précédemment décrites, le temps d'exécution est basé sur le logiciel. Par exemple dans [Kontagora & Gonzalez-Velez 2010] la commande Unix « top » est utilisée. La précision logicielle est effectivement adéquate pour comparer des hyperviseurs entre eux ou avec des systèmes non virtualisés. Cependant, cette précision n'est pas probante pour comparer une machine virtuelle avec un hyperviseur ou un système non virtualisé. En effet, comme le montre la Figure IV.5 un hyperviseur peut modifier l'horloge de la machine virtuelle pour masquer le fait qu'elle est virtualisée.

De plus, afin d'évaluer précisément le temps d'exécution sur machine virtuelle, une commande logicielle telle que « *time* » n'est pas fiable à cause du comportement de l'hyperviseur.

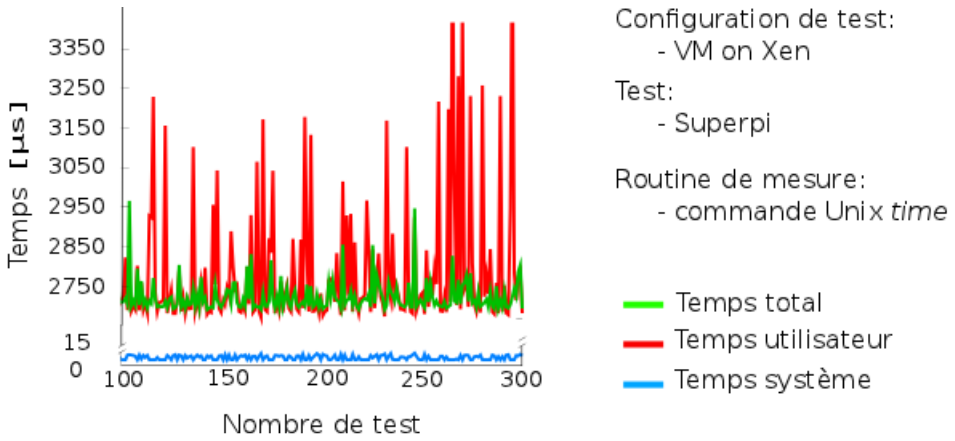

Figure IV.5 – Exemple de mesure du temps d'exécution par outil logiciel

Un exemple de mesure utilisant la commande *time* est donné par la Figure IV.5. Lors de cette expérimentation nous avons utilisé le programme « superpi » qui calcule le chiffre π sur plusieurs décimales. Notre configuration utilise l'hyperviseur Xen avec une machine virtuelle exécutant un système d'exploitation fedora 13. La commande *time* renvoie trois temps d'exécution :
le temps système est le temps passé par le processus dans l'espace noyau décrit à la Section IV.1.2, le temps utilisateur, qui correspond au temps passé dans l'espace utilisateur et le temps total qui correspond à la somme de ces deux temps.

Or, nous remarquons un comportement aberrant. En effet, le temps que passe le processus dans l'espace utilisateur est supérieur au temps total. Ce comportement peut s'expliquer par le fait que l'outil logiciel utilisé n'a pas été spécifiquement développé pour une utilisation sur une machine virtuelle. Par conséquent, la prise en charge des différentes couches de virtualisation mise en jeu peut altérer les mesures.

De plus, de nombreuses horloges sont présentes dans un ordinateur et différents hyper-

viseurs peuvent utiliser les horloges de leur choix. Afin de connaitre l'horloge utilisée, un accès au code source de l'hyperviseur doit être fait, tous ne sont pas librement accessibles.

Par conséquent, nous avons besoin d'une nouvelle méthode de mesure du temps d'exécution qui doit être aussi indépendante que possible de la cible logicielle visée. La suite de cette section présente notre deuxième contribution [Lastera *et al.* 2011], qui implémente une telle approche.

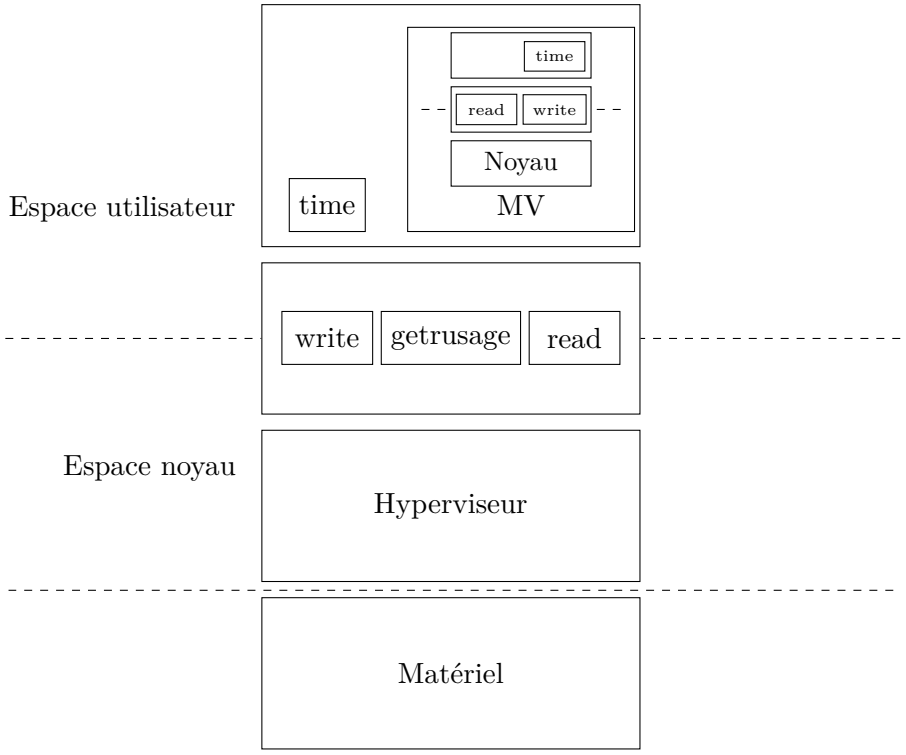

Figure IV.6 – Vue des différents niveaux d'un système virtualisé

IV.2.1 Critères considérés pour la présélection des hyperviseurs

Nous rappelons que cette comparaison des solutions de virtualisation doit permettre de sélectionner un hyperviseur afin d'être déployé sur l'architecture logicielle présentée au Chapitre II Section II.4 et mettre en œuvre également le mécanisme de détection présenté au Chapitre III Section III.3.

Afin de comparer différentes solutions de virtualisation, nous avons défini les critères de sélection correspondants aux besoins du domaine avionique auquel correspond notre étude. Tout d'abord nous nous focalisons sur des solutions de type virtualisation native, où comme expliqué précédemment, l'hyperviseur s'exécute directement au dessus de la couche matérielle. Les hyperviseurs doivent supporter les processeurs de la famille Intel x86, ce type de processeur étant dominant sur le marché des ordinateurs portables grand public que nous ciblons dans notre étude. De plus il est primordial, afin d'assurer une diversification de l'environnement d'exécution de l'application de maintenance, d'utiliser au minimum deux systèmes d'exploitation différents. Le support du système d'exploitation Windows est obligatoire afin de correspondre au demande des compagnies aériennes. Le

deuxième système utilisé pouvant être un système de type Unix.

Bien que les opérations de maintenances ne soient pas soumises aux mêmes exigences de certification que les opérations de vol, il serait souhaitable d'anticiper une telle demande par l'utilisation d'hyperviseurs ayant obtenu des certifications [1]. Le dernier critère à considérer est la disponibilité d'utilisation des hyperviseurs. En effet, de nombreuse solutions de virtualisation étant sous licence propriétaire payante, et destiné principalement aux industriels, les tarifs prohibitifs ainsi que les négociations avortées avec certains éditeurs nous ont conduit à placer le critère de disponibilité en premier lieu afin de pouvoir réaliser notre étude.

Le Tableau IV.1 présente les différentes solutions de virtualisation que nous avons présélectionnées afin d'obtenir une liste d'hyperviseurs en adéquation avec les critères retenus. Notre présélection à pris en compte à la fois des solutions majeures du marché de la virtualisation tel que VMware et des projets de recherche tel que Nova.

Parmi les six hyperviseurs disponibles pour l'étude, nous avons retenu l'hyperviseur Xen car, en plus des critères requis, il s'appuie sur une large communauté d'utilisateurs et jouit d'une bonne réputation. Qubes-OS a attiré notre attention du fait de son orientation prononcée pour la sécurité. La solution VMware ESXi n'a pas été retenue car elle n'est pas adaptée à un environnement portable. Les solutions XtratuM et Nova ont été rejetées car elles ne répondent pas aux critères de support du système d'exploitation Windows. L'hyperviseur KVM à également été rejeté car il permet d'utiliser uniquement la technique de virtualisation complète.

1. Dans le cadre de notre étude seules les certifications selon les critères communs présentées au Chapitre I Section I.3.3 sont retenues

Caractéristiques \ Hyperviseurs	Enea	Green Hills	KVM	LynxSecure	NOVA	OKL4	PikeOS	Polyxène	Qubes-OS	RTS Hypervisor	VMware ESXi Server 3.5	WindRiver	Xen	XtratuM
Certification	Non	EAL6+	Non	Non	Non	Non	Non	EAL5	Non	Non	EAL4+	Non	Non	Non
Disponibilité	Non	Non	Oui	Non	Oui	Non	Non	Non	Oui	Non	Oui	Non	Oui	Oui
Famille processeur	Freescale QorlQ	ARM Intel Power	ARM Intel PowerPC s390	Intel	Intel	ARM Mips Intel	Intel PowerPC AMCC SPARC LEON MIPS ARM SH-4	Intel	Intel	Intel	Intel	Intel PowerPC	Intel PowerPC ARM	Leon2 Intel
OS supportée par l'hyperviseur	Linux RTOS Enea OSE	Linux Windows Solaris Vxworks	Linux Windows Unix system	Linux Windows LynxOS SE Linux	Multiple OS non modifié	Linux Android	Linux Legacy RTOS RTEMS	Polyxène OS Windows Linux	Windows Linux	Windows XP Windows CE VxWorks RTOS-32 QNX OS-9 Linux real-time Linux	Windows Linux Solaris Novell NetWare	VxWorks windriver Linux	Windows Linux Solaris BSD	Rtems PaRTiKle Linux

Tableau IV.1 – Comparatif des solutions de virtualisation présélectionnées

IV.2.2 Présentation des hyperviseurs sélectionnés

Xen

Le projet Xen a été initialement développé au sein du laboratoire informatique de l'université de Cambridge par le *Systems Research Group* (SRG) [Barham *et al.* 2003] en 2003.

Xen est un hyperviseur *open source* supportant les processeurs 32 et 64 bits, il permet la virtualisation complète ainsi que la virtualisation native. Lors de notre comparaison, seule la virtualisation native a été utilisée. La couche logicielle utilisée par Xen se compose de deux domaines distincts comme l'illustre la Figure IV.7 : le domaine 0 (Dom0) et le domaine U (DomU).

Le Dom0 est un domaine privilégié exécuté en premier lorsque le système démarre. Le terme DomU (*unprivileged*) représente les domaines non privilégiés .

Le Dom0 a un accès direct au matériel et permet de créer et de configurer des DomU ainsi que de leur fournir une classe générique de périphérique accessible. Par exemple, une carte réseau est vue comme un périphérique réseau générique par le DomU. Le Dom0 est le seul à avoir les pilotes nécessaires afin d'assurer la communication avec les périphériques.

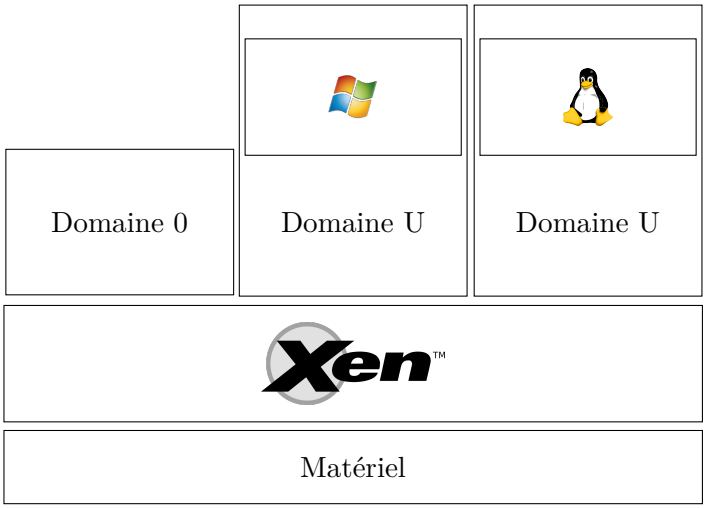

Figure IV.7 – Architecture de Xen

Qubes

Qubes est un système d'exploitation sécurisé pour les ordinateurs grand public basé sur l'hyperviseur Xen [Rutkowska 2010]. Il se démarque de ce dernier par l'utilisation de domaine particulier pour la gestion du réseau, du stockage ainsi que pour la gestion des différents systèmes invités. La Figure IV.8 présente l'architecture globale de Qubes. Nous remarquons que contrairement à Xen la gestion du réseau et du stockage est placée dans un domaine non privilégié. Le domaine 0 ne contient que les processus d'administration du système ainsi qu'une interface graphique sécurisée.

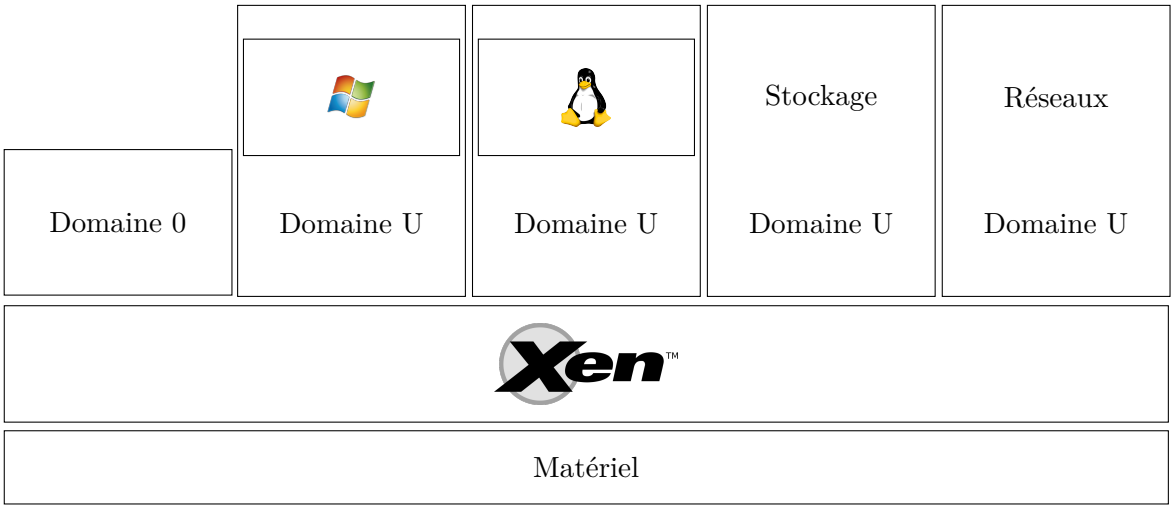

Figure IV.8 – Architecture de Qubes

IV.2.3 Évaluation

Notre approche diffère des différentes études citées précédemment par l'utilisation de matériel indépendant de la configuration ciblée pour l'évaluation. Cela nous permet à la fois de réaliser les mesures de manière précise et d'observer de façon détaillée le comportement de la configuration.

Technique de mesure

La Figure IV.9 décrit de manière schématique l'outil de mesure, qui comprend l'architecture Intel x86 de l'ordinateur utilisé pour les tests ainsi qu'un oscilloscope. Pour nos programmes de tests, afin de maitriser leur comportement et de limiter les couches d'abstraction avec le matériel, le langage assembleur est privilégié. L'instruction `out` est utilisée par nos programmes de test de manière à envoyer des données sur le port parallèle. Le processeur exécute l'instruction et active des lignes dédiées sur le *Front-Side Bus* (FSB). Le *Northbridge* convertit les signaux en requêtes PCI (*Peripheral Component Interconnect*), qui sont acheminées par la suite vers le *Southbridge*. Enfin, le *Southbridge* convertit ces requêtes en requêtes LPC (*Low Pin Count*), qui ensuite activent ou désactivent un bit du port parallèle. Un oscilloscope est relié à ce bit afin d'observer le comportement de la configuration ciblée et de stocker les résultats pour analyse.

Un environnement spécifique a été développé pour mesurer le temps d'exécution d'un ensemble de programmes de test conçu de manière a obtenir une charge de travail élevée sur des composants matériels : mémoire et processeur.

Critères d'évaluation

Les mesures recueillies visent à fournir des informations sur la possible surcharge imputée à l'utilisation de la technologie de virtualisation.

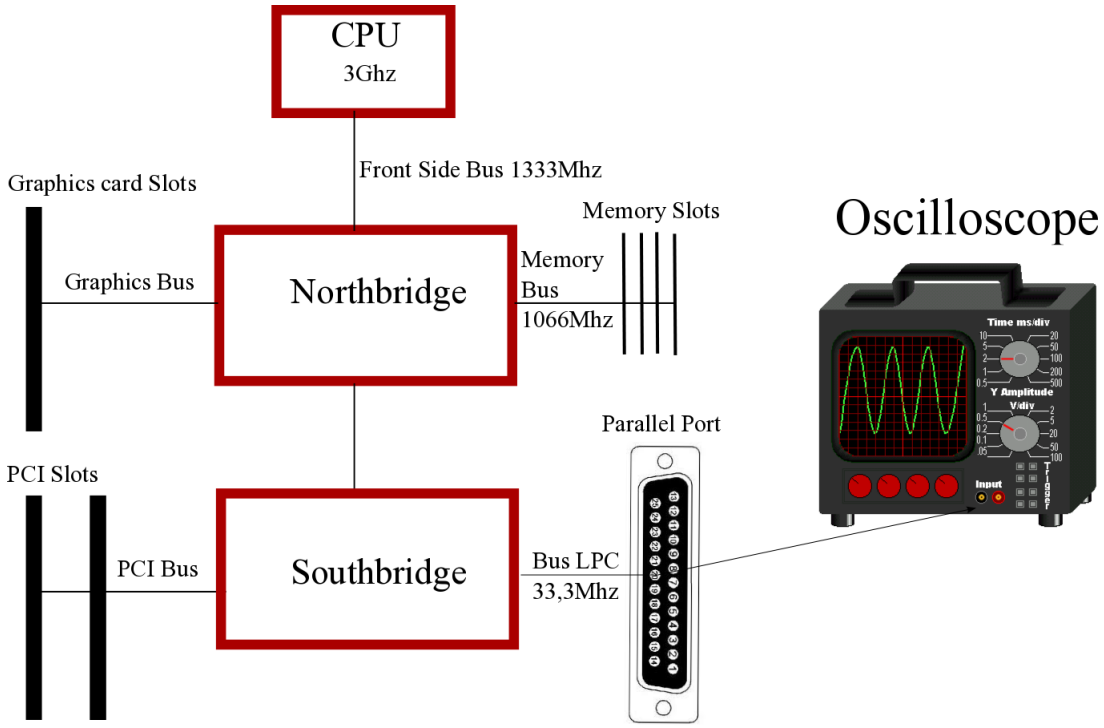

Figure IV.9 – Schéma de l'outil de mesure

Les programmes de test sont exécutés sur sept configurations différentes, décrites sur la Figure IV.10.

L'analyse de ces sept configurations vise a fournir une meilleure compréhension de l'impact sur les performances des différentes couches (matériel, OS, hyperviseur, machine virtuelle).

Tout d'abord, nous considérons une configuration minimale nommée « Pépin », qui fournit un accès direct au matériel et ne présente aucune optimisation de gestion mémoire, CPU ou réseaux. Ensuite, nous considérons deux version du noyau Linux (2.6.32 et 2.6.34). Ceci à pour but d'affiner notre analyse sur la performance des deux hyperviseurs : Xen basé sur le noyau 2.6.32 et Qubes basé sur le noyau 2.6.34. Cette différence de version du noyau est due au date différente de lancement des projets. En effet Xen a débuté en 2003 avec une version du noyau 2.6 qui a évolué au cours du temps pour atteindre la version 2.6.32 au début de notre étude. En ce qui concerne le projet Qubes, débuté en 2010, son développement a commencé avec la version du noyau 2.6.34. Enfin, nous considérons deux configuration avec une machine virtuelle et un système d'exploitation appelé « Xen-vm »et « Qubes-vm ». Dans ces deux configurations la machine virtuelle prend en charge le système d'exploitation Fedora 13.

Calibration de l'outil de mesure

Notre outil de mesure doit d'abord être calibré afin d'évaluer le biais imputé au temps de traversée des différentes couches des configurations. À cette fin, nous avons développé un programme d'étalonnage appelé Burst. Son principe est simple : il envoie des niveaux bas et haut sur une broche du port parallèle afin d'émettre un signal carré (Figure IV.11). La

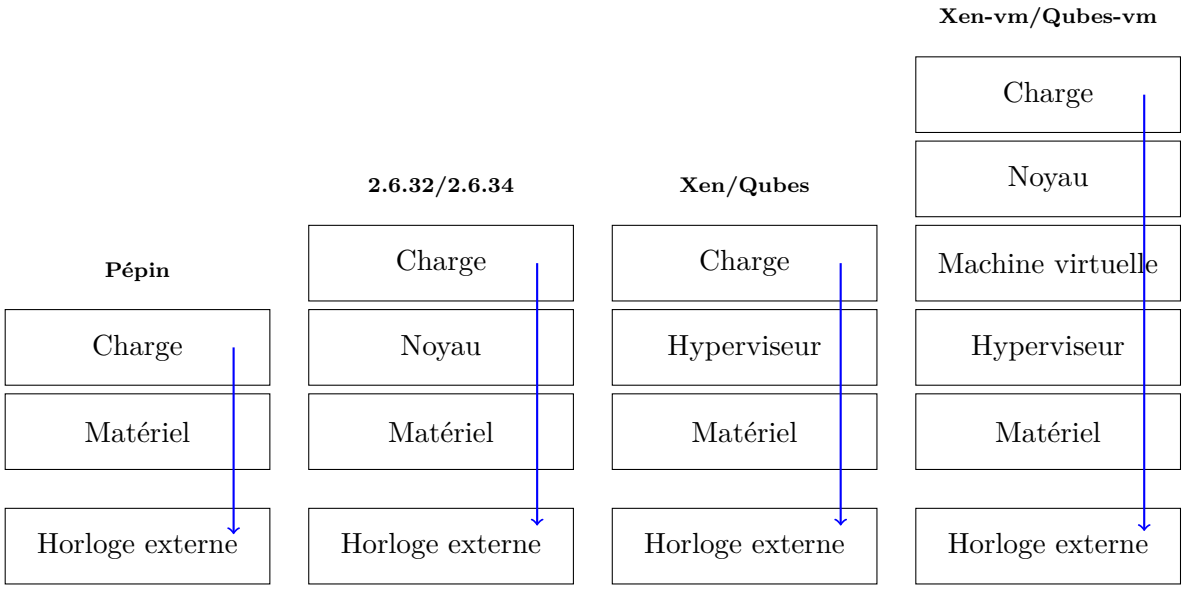

Figure IV.10 – Configurations étudiées

demi-période de l'onde carrée, telle que mesurée sur l'oscilloscope, est le temps nécessaire pour traverser les différentes couches logicielles entre le programme et le matériel.

Algorithm 1 Pseudo Code - Test Burst

```
while true do
  ecrire 0 sur le port
  ecrire 1 sur le port
end while
```

Figure IV.11 – Algorithme du test de calibration

La Figure IV.12 représente le temps moyen en microsecondes afin qu'une instruction d'écriture traverse les différentes couches de chaque configuration. Les intervalles de confiance de 95% ont été calculés pour évaluer la pertinence des données. Les configuration « 2.6.32 » et « 2.6.34 » fournissent le meilleur résultat avec un temps de traversée de 1,27 microsecondes. La configuration « Pépin » obtient un temps inférieur par son absence d'optimisation. La configuration « Xen » est la plus lente pour ce test d'étalonnage. Nous notons cependant que seulement une différence d'environ 600 ns est mesurée entre ces deux configurations. Cela représente une erreur de mesure potentielle de moins de 1% alors que notre outil de mesure est utilisé afin d'évaluer la performance d'un programme avec un temps d'exécution supérieur à 60 microsecondes.

Programmes de tests

Nous souhaitons évaluer la dégradation causée par l'hyperviseur sur les performances de

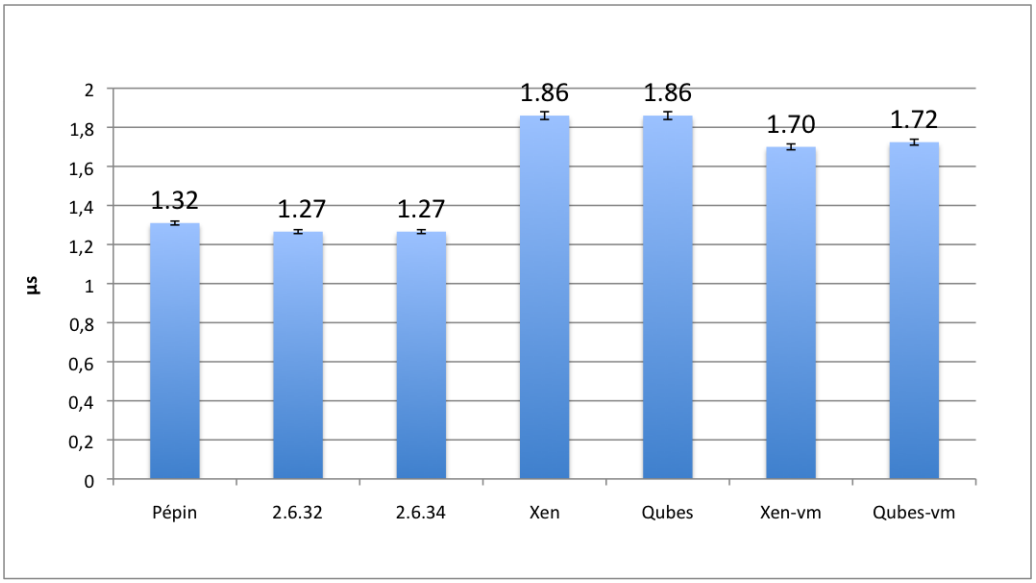

Figure IV.12 – Temps moyen mesuré pour la traversée des différentes couches : Programme de calibration

plusieurs types d'applications qui peuvent être déployées dans un environnement avionique. Par conséquent, nous avons développé des programmes types qui sollicitent chacun un composant particulier du matériel (CPU, mémoire et réseaux). Un programme complexe exploite ces ressources matérielles à des degrés différents, de sorte que la dégradation des performances peut être considérée comme une somme pondérée de la dégradation des performances mesurées par ces tests de base.

Les applications sont développées de la manière suivante : écriture de zéro sur un bit du port parallèle avant le lancement de l'application puis écriture de un sur le même bit du port parallèle à la fin de l'exécution. Ainsi, le temps qui s'écoule sur le palier bas mesuré à l'oscilloscope représente le temps d'exécution de la charge.

Test processeur

Le test CPU doit permettre d'identifier si l'hyperviseur exploite pleinement les capacités du processeur. L'instruction de multiplication `imul` est utilisée. La taille de l'échantillon de l'étude a été fixée à mille expériences.

Algorithm 2 Pseudo Code - Test Processeur

```
while true do
  ecrire 0 sur le port
  multiplication de C par D
  ecrire 1 sur le port
end while
```

Figure IV.13 – Algorithme du test processeur

Test mémoire

Le test mémoire doit permettre d'identifier si l'hyperviseur exploite pleinement les capacités du bus FSB (*Front-Side Bus*). Ce dernier permet de connecter le processeur à la mémoire. Le programme de test de mémoire, présenté à la Figure IV.14 est écrit en langage assembleur pour nous permettre de contrôler les instructions exactes exécutées par le processeur.

Algorithm 3 Pseudo Code - Test Copie

```
while true do
  ecrire 0 sur le port
  copie de la table A vers la table B
  ecrire 1 sur le port
end while
```

Figure IV.14 – Algorithme du test mémoire

La dimension des tables mémoire A et B est fixée à 10 Mo. L'opération est répétée dix fois afin de saturer les différents caches matériel. Le total des données copiées est donc de 100Mo. La taille de l'échantillon de l'étude a été fixée à mille expériences.

Test réseau

Nous utilisons une configuration client/serveur pour évaluer les performances du réseau entre l'hyperviseur et la machine virtuelle. Le programme de test se compose de deux parties : la première présentée en Figure IV.15 est utilisée par l'hyperviseur comme application serveur, la seconde partie, l'application client, permet à la machine virtuelle d'envoyer les messages. La taille des messages a été définie à 10Mo. L'application client envoie le message 100 fois.

Algorithm 4 Pseudo Code - Test Réseau

```
ecrire 0 sur le port
while taille.message <>0 do
  message = message reçue depuis la MV
end while
ecrire 1 sur le port
```

Figure IV.15 – Algorithme du test réseau

IV.2.4 Analyse des résultats

CPU

La Figure IV.16 montre les valeurs moyennes sur mille expériences pour chaque configuration. L'échelle de temps est en millisecondes. Les intervalles de confiance à 95% ont été calculés pour mesurer la pertinence des données.

Nous observons que les configurations « 2.6.32 » et « 2.6.34 » obtiennent la performance la plus faible en raison de différentes routines noyaux indispensables aux systèmes. Si l'on considère les cinq configurations restantes, la configuration « Pépin » est la plus lente, en raison de son minimalisme et de l'absence d'optimisation. « Qubes » obtient le meilleur résultat pour ce test. Les configurations « Xen », « Xen-vm » et « Qubes-vm »ont un comportement similaire si l'on tient compte de l'intervalle de confiance.

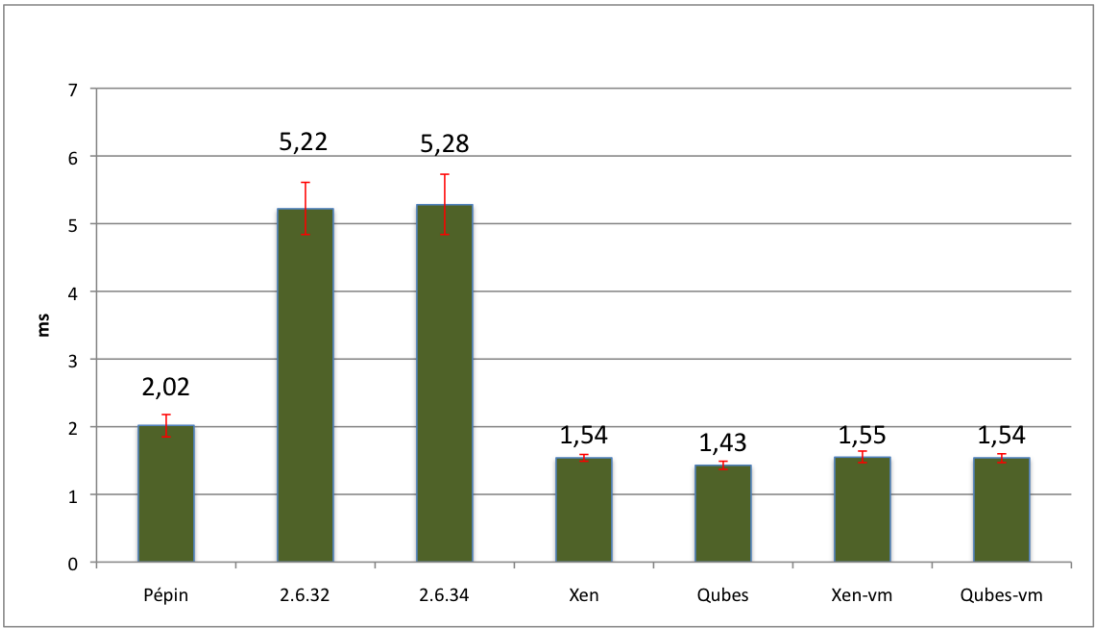

Figure IV.16 – Temps moyen d'exécution du programme CPU

Mémoire

Les histogrammes de la Figure IV.17 montrent les valeurs moyennes pour mille expériences sur chaque configuration. Les intervalles de confiance à 95% ont été calculés pour mesurer la pertinence des données. L'échelle du temps est en millisecondes. Nous observons que la configuration « Pépin » n'est pas optimisée, en fait elle a été conçue sans critères de performance à l'esprit. Le seul but de cette configuration est d'envoyer des données via le port parallèle.

Au contraire, il est visible que les systèmes d'exploitation et les hyperviseurs peuvent être optimisés pour améliorer les performances d'entrées/sorties. La différence entre les deux versions du noyaux linux peut être expliquée par les améliorations introduites à chaque changement de version. En ce qui concerne l'hyperviseur, « Xen » a un comportement similaire à « Pépin »ce qui suggère qu'il ne met pas en œuvre une mise en cache pour améliorer les transferts de mémoire : il a été conçu pour assurer la mise en œuvre de technique de virtualisation et d'isolement plutôt que d'être optimisé pour ses propres performances. En effet, la performance de la configuration « Xen-vm » est supérieure à la configuration « 2.6.32 » sur laquelle elle est basée « Qubes » semble suivre une philosophie différente. En effet un mécanisme de mise en cache à probablement été utilisé afin d'améliorer le transfert mémoire, et de ce fait obtenir le meilleur résultat pour ce test. Cependant, la gestion de sa machine virtuelle est moins efficace que « Xen ». Effectivement pour le test considéré, « Qubes-vm » est moins performante que « Xen-vm ».

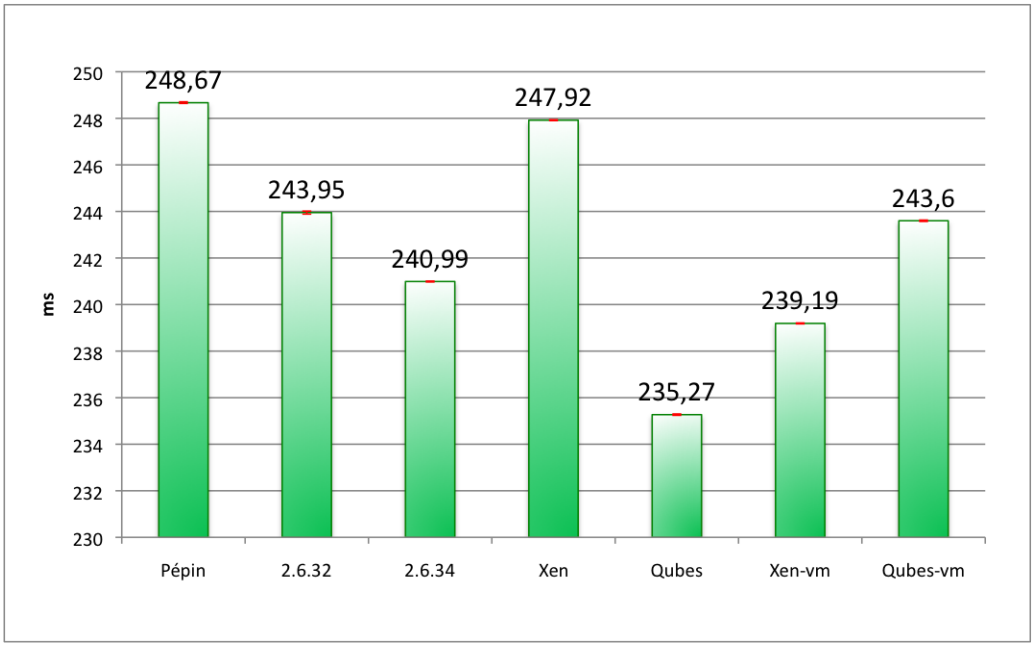

Figure IV.17 – Temps moyen pour le transfert de 100Mb de mémoire à mémoire

Réseaux

Les histogrammes de la Figure IV.18 montrent les valeurs moyennes pour cent expériences sur chaque configuration. Les intervalles de confiance à 95% ont été calculés

pour mesurer la pertinence des données. L'échelle du temps est en millisecondes. Les performances réseaux des deux hyperviseurs sont sensiblement identiques compte tenu de l'intervalle de confiance.

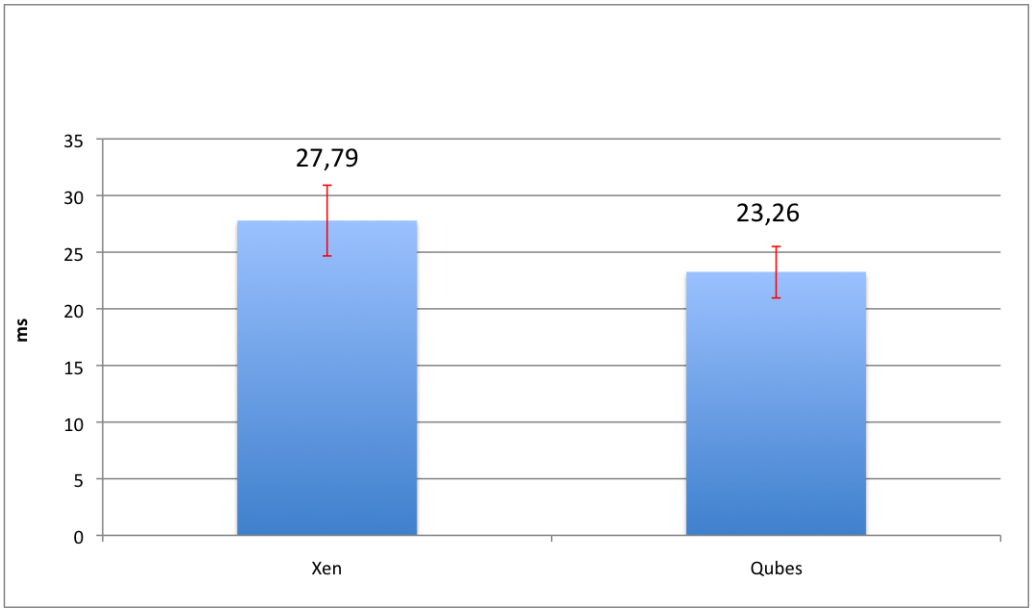

Figure IV.18 – Temps moyen d'exécution du test réseau

Conclusion

Ce chapitre a présenté une méthodologie d'évaluation des performances d'une couche de virtualisation. En effet, nous nous intéressons à la dégradation induite par l'utilisation d'un hyperviseur. Nous avons montré que l'utilisation d'un accès matériel tel que le port parallèle permet d'être indépendant des configurations à évaluer.

Pour ce faire, nous avons présenté les différents types de virtualisation avant de détailler les éléments techniques. Par la suite, nous avons présenté les différents critères retenus pour sélectionner les hyperviseurs et permettre leur évaluation. Du fait de la difficulté d'obtention d'hyperviseurs propriétaire, notre choix s'est tourné vers deux solutions *open source* (Xen et Qubes). Nous avons ensuite décrit notre banc de test ainsi que les différents programmes utilisés pour évaluer les hyperviseurs. L'analyse des tests que nous proposons montre la faible surcharge engendrée par l'utilisation d'une solution de virtualisation.

Suite à cette évaluation, l'hyperviseur Xen à été choisit comme couche de virtualisation afin de mettre en œuvre l'architecture présentée dans le deuxième chapitre (Section II.4). La présentation du démonstrateur dans le chapitre suivant nous permet de détailler son utilisation.

Chapitre V

Conception et développement

Introduction

Ce chapitre à pour objectif de présenter le démonstrateur que nous avons développé pour prendre en charge les opérations de maintenance, tout en assurant des propriétés de sécurité, à la fois au sens de l'immunité et de l'innocuité. Ce démonstrateur concerne le cas d'étude présenté au chapitre II utilisant un portable de maintenance (PMAT) et implémente les deux mécanismes décrit au chapitre III.

Dans un premier temps, nous détaillons l'architecture logicielle du PMAT, puis nous poursuivons par la présentation de l'application de maintenance, de sa conception à son utilisation. Dans un deuxième temps, nous présentons une première expérimentation du démonstrateur avec une application simplifiée. Nous conclurons ce chapitre par l'expérimentation de l'application réelle de maintenance.

V.1 Contexte d'utilisation

Le PMAT doit permettre à l'opérateur de réaliser des opérations de maintenance de manière sécurisée tout en apportant une souplesse d'utilisation ainsi qu'une grande mobilité.

Par exemple, l'opérateur doit pouvoir accéder depuis son bureau à sa messagerie professionnelle afin de consulter son planning, mais également effectuer différentes tâches administratives comme la rédaction d'un rapport de maintenance. L'accès aux sites des constructeurs avec pour objectif d'assurer une veille technologique doit également être permis. Ces opérations doivent être possibles sur le lieu de travail de l'entreprise en charge de la maintenance et également hors du site de l'entreprise lorsque l'opérateur est en déplacement.

La réalisation de ces différentes opérations en plus des tâches de maintenance ne peut pas être effectuée sur un même ordinateur sans l'utilisation d'une architecture sécurisée

spécifique. Nous détaillons dans la suite de cette section l'architecture logicielle du PMAT.

V.1.1 Détail de l'architecture logicielle du PMAT

La Figure V.1 reprend l'architecture logicielle décrite au chapitre II (Figure II.5) en détaillant les composants logiciels utilisés. Tout d'abord, suite à l'évaluation que nous avons faite des différents hyperviseurs au chapitre IV, nous avons retenu l'hyperviseur Xen, pour des questions de maturité et du fait de sa large utilisation. En effet, Qubes-OS est encore en cours de développement et souffre d'une communauté d'utilisateurs nettement moins importante que celle de Xen. De plus, il se destine principalement à une utilisation bureautique et non à une utilisation embarquée dans le domaine aéronautique. L'hyperviseur héberge, en plus des machines virtuelles, l'objet de validation, composé des deux mécanismes de comparaison désignés sur la Figure V.1 par CmpA pour le mécanisme utilisant la comparaison des traces et par CmpB pour le mécanisme utilisant le modèle d'exécution.

Afin d'assurer la diversification de l'environnement d'exécution de l'application de maintenance, nous utilisons deux systèmes d'exploitation : Windows et Linux. Le choix du système Windows repose sur le fait de sa large utilisation par les compagnies aériennes depuis de nombreuses années. L'utilisation d'un système de type Linux repose sur son utilisation de plus en plus significative dans l'industrie. Un système d'exploitation tel que Mac OS n'a pas été retenu afin de ne pas limiter le choix de l'ordinateur portable au seules machines Apple. En effet, selon le contrat de licence de logiciel MAC OS X [APPLE 2011], « Les droits accordés par la présente Licence ne vous autorisent pas à installer, utiliser ou exécuter le logiciel Apple sur un ordinateur qui ne soit pas de marque Apple ».

Une machine virtuelle Java (JVM) doit être installée sur les deux systèmes d'exploitation afin d'assurer l'exécution de l'application de maintenance.

V.1.2 Description et utilisation de l'application de maintenance

L'application mise à notre disposition par Airbus, est une version de développement. En plus du profil Maintenance elle comporte les profils Pilot et Administrator que nous n'utiliseront pas car ils ne sont généralement pas implémentés sur un PMAT.

Le profil maintenance permet de réaliser des opérations parmi lesquelles l'enregistrement des erreurs et alertes survenues durant le vol, et les actions de test.

L'enregistrement des erreurs et alertes survenue durant le vol ou *Post Flight Report*(PFR) permet à l'opérateur de maintenance de connaître les différentes opérations de maintenance qu'il doit effectuer.

Les actions de test permettent de s'assurer de la bonne réalisation des opérations de maintenance.

Nous présentons sur la figure V.2 les différentes étapes liées à la conception, au développement et au déploiement de l'application de maintenance sur l'architecture diversifiée pro-

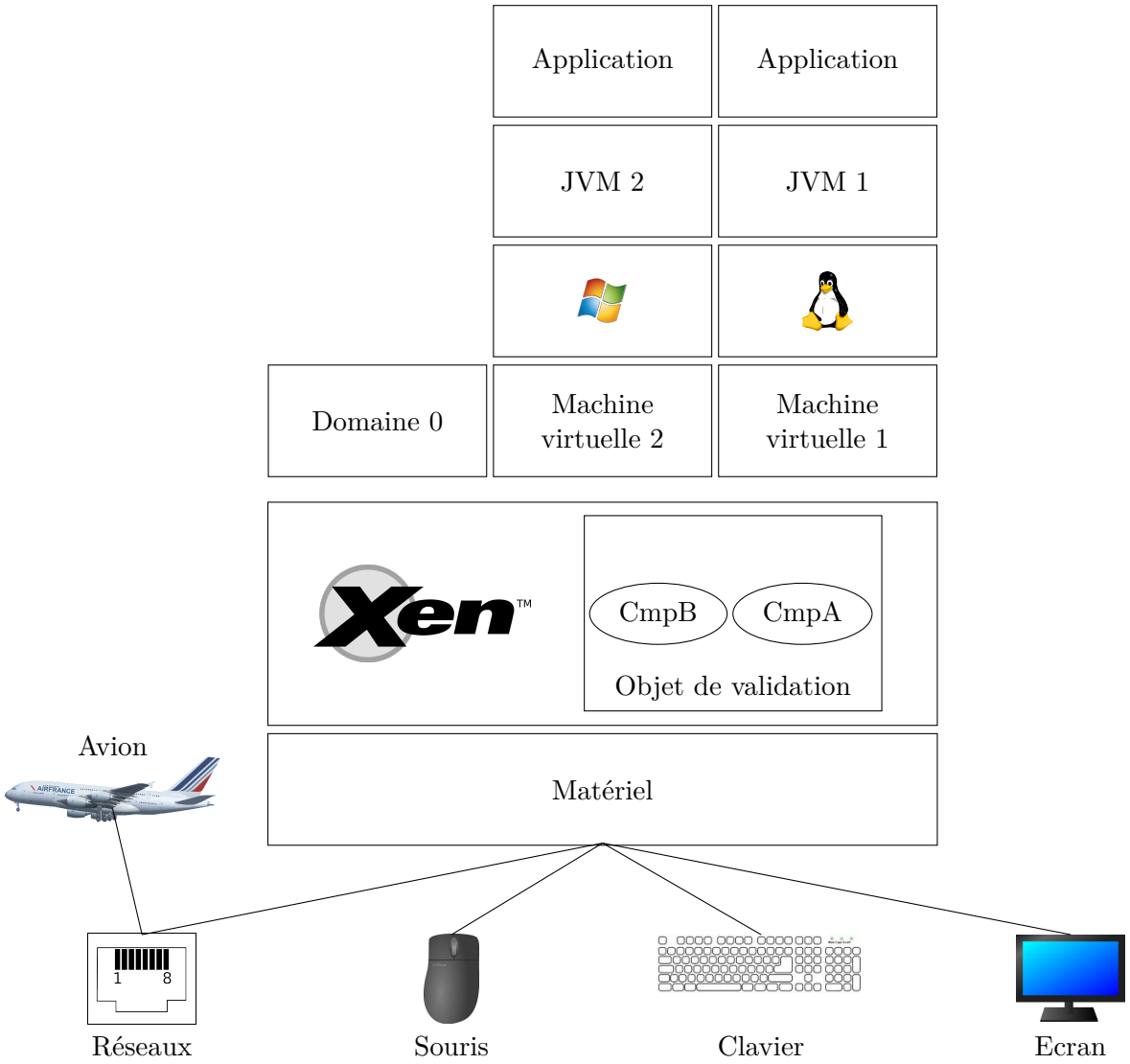

Figure V.1 – Architecture logicielle du PMAT

posée dans notre étude.

La première opération (①) est réalisée par le bureau d'études et consiste notamment au développement de l'application de maintenance en respectant les contraintes liées au domaine avionique. Cette application est écrite en langage Java pour assurer la portabilité sur différentes cibles (PC, Notebook, Tablette numérique). Après validation de l'application de maintenance, elle est téléchargée de manière sécurisée sur l'avion (②). La troisième opération (③) représente le téléchargement de manière sécurisée du binaire de l'application sur l'ordinateur lors d'une opération de maintenance. La quatrième opération (④) correspond au téléchargement de l'application de maintenance sur les deux systèmes d'exploitation présents sur l'ordinateur portable de maintenance.
La cinquième opération (⑤) concerne les mécanismes de comparaison. Tout d'abord nous rappelons que la comparaison porte sur les appels graphiques de l'application de maintenance. Tous les événements liés à la bibliothèque graphique (Swing) constituent les traces d'exécution. Le mécanisme CmpA vérifie les traces d'exécution des deux instances : les événements sont contrôlés, un par un, après réception. Si un événement est manquant, un

délai est alors considéré afin de tolérer les éventuels décalages temporels. Si le délai imparti est dépassé ou que les événements ne sont pas les mêmes ceci indique une divergence dans les exécutions et les traces sont déclarées invalides. Ensuite, le mécanisme CmpB analyse ces mêmes traces d'exécution une à une, en les opposant au modèle d'exécution de l'application. Si une trace d'exécution ne correspond pas au modèle, elle est déclarée invalide. Enfin, si les traces d'exécution sont déclarées invalides par les deux mécanismes, cela indique un dysfonctionnement ou une malveillance au niveau des machines virtuelles. L'exécution est alors arrêtée et une alerte est levée.

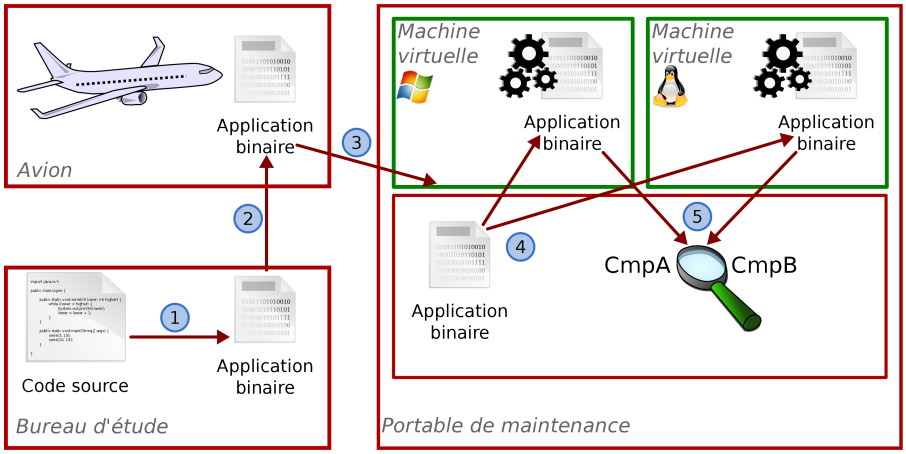

Figure V.2 – Diagramme opérationnel du déploiement de l'application sur le PMAT

V.1.3 Installation et expérimentation du démonstrateur GEODESIE

L'installation et l'exécution du démonstrateur s'effectuent en 4 étapes.

Premièrement, le bytecode Java de l'application que nous souhaitons étudier est placé dans un répertoire de l'hyperviseur. L'analyse du bytecode Java s'effectue ensuite en deux étapes dans des répertoires différents. Enfin, l'exécution du démonstrateur permet à l'hyperviseur de télécharger l'application sur les machines virtuelles et de lancer leur exécution. Dans la suite de cette section, nous détaillons l'arborescence du programme d'analyse réalisé en langage Java, puis nous décrivons les principales étapes d'installation.

Avant de détailler l'installation du démonstrateur nous débutons, la suite de cette section par une description de l'arborescence de notre programme afin de faciliter la compréhension des différentes étapes d'installation.

Installation

Nous utilisons six répertoires afin de séparer les différents fichiers. Le répertoire `Classes` contient le bytecode Java de l'application d'analyse. Le bytecode Java de l'application à

analyser est enregistré après injection d'instructions dans le répertoire `Injected`. Les librairies indispensables au bon fonctionnement de l'application sont stockées dans le répertoire `Library`. Le répertoire `Script` contient les différentes commandes à exécuter. Le code Java de l'application effectuant l'analyse est stocké dans le répertoire `Sources`. Le répertoire `Outputs` contient les fichiers de log des différentes commandes contenues dans le répertoire `Script`. Dans la suite de cette section, nous détaillons les différentes commandes contenues dans le répertoire `Script` en mettant l'accent sur les commandes d'installation.

Compilation

```
javac -classpath Library -d Classes Sources > Outputs/out_compiler
```

La commande `javac` assure la compilation du programme à partir des fichiers contenus dans le répertoire `Sources`. Le bytecode Java est généré dans le répertoire `Classes`. L'option *-classpath* permet l'utilisation de librairies contenues dans le répertoire `Library`. Les sorties de la commande `javac` sont redirigées vers le fichier `out_compiler`.

Modélisation

```
java Monitor.main -A --bytecode Target --model Model > Output/out_modeler
```

La commande `java` exécute la classe `Monitor.main` sur le bytecode Java de l'application. Les sorties de la commande `java` sont redirigées dans le fichier `out_modeler`.

Cette étape nous permet d'identifier les différents appels graphiques et les différents échanges réseaux que nous souhaitons intercepter.

Injection

```
java Monitor.main -I --bytecode Injected --model Model > Output/out_injector
```

La commande `java` exécute la classe `Monitor.main` sur le bytecode Java de l'application obtenue après l'étape de modélisation. Les sorties de la commande `java` sont redirigées dans le fichier `out_injector`.

Cette étape nous permet d'injecter du bytecode Java autour des appels que nous avons identifiés lors de l'étape précédente.

Exécution

```
java Monitor.main -S --model Model --port 1234> Output/out_supervisor
```

La commande `java` exécute la classe `Monitor.main` sur le bytecode Java de l'application obtenue après l'étape d'injection. Les sorties de la commande Java sont redirigées dans le fichier `out_supervisor`. Cette étape permet l'exécution du superviseur.

```
Java com.sii.maintenanceSystem.core.LoginPage > Output/out_executor
```

La commande `java` exécute la classe `LoginPage`. Les sorties de la commande Java sont redirigées dans le fichier `out_executor`. Cette étape permet l'exécution de l'application de maintenance.

Expérimentation sur une application simplifiée

Cette expérimentation a pour but de présenter l'utilisation de notre méthode sur un programme simple. Dans notre cas, il s'agit de la création d'une *frame* avec différents composants comme la taille, le titre ou la visibilité. Les paramètres de ces composants peuvent être modifiés par une personne malveillante afin d'altérer la *frame*. Pour éviter un tel acte, nous devons intercepter les quatre appels graphiques concernés :

- La création d'une frame par l'appel de la méthode new()
- La définition d'un titre par l'appel de la méthode setTitle()
- La définition de la taille par l'appel de la méthode setSize()
- La définition de la visibilité par l'appel de la méthode setVisible()

Après la réalisation de l'étape de modélisation décrite précédemment, nous obtenons le fichier `out_modeler` présenté dans le Tableau V.1 comprenant les appels graphiques à intercepter.

```
Analyze: bytecode=../TARGET/bin model=outputs/model
INTERCEPT 0: InstructionMethodInsn 183: swing/JFrame.<init>()V
INTERCEPT 1: InstructionMethodInsn 182: swing/JFrame.setTitle(String;)V
INTERCEPT 2: InstructionMethodInsn 182: swing/JFrame.setSize(II)V
INTERCEPT 3: InstructionMethodInsn 182: swing/JFrame.setVisible(Z)V
```

Tableau V.1 – Extrait du fichier `out_modeler`

Après l'identification des quatre appels graphiques présentés précédemment, l'étape suivante concerne l'injection de bytecode Java afin d'intercepter les méthodes identifiées. Un extrait du fichier `out_injector` est présenté dans le Tableau V.2.

Le Tableau V.3 présente le bytecode du cas d'étude simplifié avant et après injection.

Nous représentons en bleu le bytecode Java injecté autour de l'appel de la méthode new. Cette injection permet d'invoquer la méthode `sendNew` qui, à son tour, invoque l'hyperviseur pour l'informer de la création d'un nouvel objet graphique.

```
Analyze: bytecode=../TEST/bin model=outputs/model
INTERCEPT 0: InstructionMethodInsn 183: swing/JFrame.<init>()V
INTERCEPT 1: InstructionMethodInsn 182: swing/JFrame.setTitle(Ljava/lang/String;)V
INTERCEPT 2: InstructionMethodInsn 182: swing/JFrame.setSize(II)V
INTERCEPT 3: InstructionMethodInsn 182: swing/JFrame.setVisible(Z)V
```

Tableau V.2 – Extrait du fichier `out_injector`

bytecode avant injection	bytecode après injection
`Compiled from "Etude.java"`	Compiled from "Etude.java"
public class Etude extends java.lang.Object{	public class Etude extends java.lang.Object{
public Etude() ;	public Etude() ;
...	...
public static void main(java.lang.String[]) ;	public static void main(java.lang.String[]) ;
Code :	Code :
0 : new #16 ; //class javax/swing/JFrame	0 : new #15 ; //class javax/swing/JFrame
3 : dup	3 : dup
4 : invokespecial #18 ;JFrame."<init>" :()V	4 : aconst_null
7 : astore_1	5 : ldc #16 ; //int 0
8 : aload_1	7 : bipush 0
...	9 : anewarray #4 ;
	12 : ldc #18 ;
	14 : invokestatic #24 ;
	17 : dup
	18 : invokespecial #25 ;JFrame."<init>" :()V
	21 : ldc #16 ; //int 0
	23 : ldc #18 ;
	25 : invokestatic #29 ;
	28 : astore_1
	29 : aload_1

Tableau V.3 – Bytecode avant et après injection

Expérimentation sur l'application de maintenance

Le démonstrateur présente l'utilisation de notre méthode sur le elogbookA380 daté du 18 février 2010.

Après la réalisation de l'étape de modélisation décrite au paragraphe V.1.3, nous obtenons le fichier `out_modeler` comprenant les appels graphiques à intercepter. Ils sont au nombre de 3277. Ce nombre étant conséquent, nous ne détaillerons pas les appels à intercepter, mais seulement le principe de fonctionnement de la méthode.

L'étape suivante consiste à injecter du bytecode java afin d'intercepter tous les appels aux méthodes identifiées précédemment. Le fichier `out_injector` présenté au paragraphe V.1.3, contient le nombre de ligne injectées dans l'application. Comme nous l'avons montré avec le cas simplifié, pour un appel graphique identifié, nous injectons du bytecode java avant et après l'appel de la fonction, de manière à intercepter la fonction mais également permettre le bon fonctionnement du programme. Par exemple, dans le cas simplifié pour une ligne interceptée, 11 lignes sont injectées. En ce qui concerne l'application de maintenance étudiée, pour les 3277 appels graphiques recensés, nous avons injecté 30044 lignes de bytecode java.

Nous remarquons le même ratio, nombre d'appels à intercepter/nombre de lignes à injecter, qui est d'environ 1 pour 10. Cela confirme que le fonctionnement de notre méthode est identique quelque soit le nombre d'appels à intercepter.

La Figure V.3 montre l'interface de maintenance de l'application comprenant différentes informations nécessaires au bon déroulement des opérations. Nous rappelons que l'application s'exécute sur deux systèmes d'exploitation différents. Cependant l'opérateur de maintenance ne voit qu'une seule interface, issue par exemple, d'une recopie d'une des deux exécutions. L'objectif de notre méthode est d'identifier les différences de comportement au niveau des appels graphiques et de les signaler à l'opérateur.

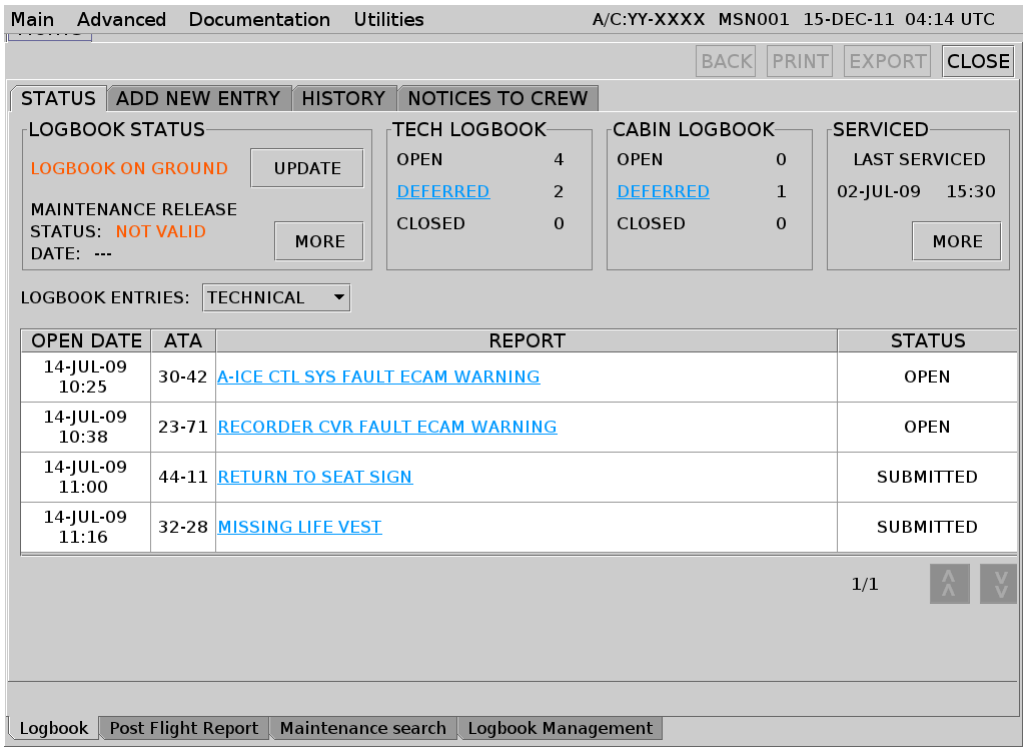

Figure V.3 – Interface graphique de la partie logbook

Dans le cas de la Figure V.3, prenons l'exemple d'un opérateur de maintenance qui souhaite mettre à jour le logbook par l'intermédiaire du bouton UPDATE. Le clic sur le

bouton UPDATE fonctionne correctement sur le système d'exploitation utilisé par l'opérateur de maintenance, mais la deuxième instance de l'application ne prend pas en compte la commande à cause d'un acte de malveillance. Comme nous l'avons décrit précédemment, tous les appels graphiques de l'application ont été recensés. Au moment de la comparaison des messages, nous serons à même d'identifier l'acte malveillant et nous pourrons le signaler à l'opérateur de maintenance.

Conclusion

Nous avons débuté ce chapitre en présentant les différents composants logiciels formant l'architecture du PMAT. Nous avons notamment décrit l'implémentation de l'objet de validation, comportant les deux mécanismes de comparaison, essentiel à notre architecture. Par la suite, nous avons décrit l'utilisation que nous faisons de l'application de maintenance de manière à illustrer notre proposition. Une hypothèse de déploiement de l'application, est également proposé, de sa création par un bureau d'étude à son exécution sur le PMAT. Cela nous permet d'avoir une vision globale du suivi de l'application et ainsi renforcer la confiance que nous avons dans l'application. Nous avons terminé ce chapitre en décrivant une expérimentation de l'application de maintenance. Bien que non exhaustive, cette expérimentation montre que notre approche peut être appliquée concrètement.

Conclusion

Ce manuscrit présente les travaux de thèse réalisés sur la définition d'une architecture de sécurité-immunité (au sens *security*) pour les équipements mobiles des systèmes avioniques.

Ils s'inscrivent dans la continuité d'une étude sur l'application des principes de redondance avec diversification pour rehausser le niveau de confiance dans l'exécution d'une fonction sur un équipement mobile.

Nous avons tout d'abord identifié les approches employées dans le domaine aéronautique pour traiter les études de sûreté de fonctionnement. En particulier, les études relatives à la sécurité-immunité et le rapprochement « sécurité-immunité/sécurité-innocuité (au sens *safety*) ». Ces études conduisent au découpage de l'avion en domaines. Certains de ces domaines permettent la connexion d'équipements mobiles. Cette ouverture des communications entraîne l'apparition de nouvelles classes de vulnérabilités. Afin de prendre en compte ces nouvelles classes de vulnérabilités, une première proposition d'architecture a été faite. Cette architecture repose sur le principe de redondance par diversification. La diversification est assurée par la technologie de virtualisation en utilisant un hyperviseur gérant deux machines virtuelles. Chacune des machines virtuelles se voit dotée d'un système d'exploitation différent. La même application est exécutée par les systèmes d'exploitation. Un mécanisme de détection est mis en place par l'hyperviseur afin de comparer les traces d'exécution de l'application redondée. Un chien de garde borne le délai de transmission des traces entre les machines virtuelles et l'hyperviseur, ainsi si les traces ne sont pas équivalentes ou si le chien de garde est dépassé, une alerte est levée, indiquant une tentative d'attaque. L'application est alors arrêtée.

Après avoir analysé cette architecture, nous avons mis en évidence l'importance d'identifier les composants logiciels permettant une mise en œuvre efficace et la nécessité d'élargir l'ensemble des classes d'attaques prises en compte.

En effet, les classes d'attaques prises en compte par l'architecture initiale ne considéraient pas les fautes de mode commun. Il était d'autant plus nécessaire d'aborder ce point que certains composants utilisés par les différentes machines virtuelles sont les mêmes. Nous avons donc développé une approche complémentaire visant à identifier des comportements de ces machines virtuelles qui ne sont pas cohérents vis-à-vis d'un modèle d'exécution. En particulier, ce modèle est construit en analysant statiquement le contenu du programme, avant son déploiement. Dans cette architecture, les fautes de mode commun se traduisent par un comportement incohérent vis-à-vis du modèle. L'ensemble des

classes de vulnérabilités prises en compte est ainsi plus grand. Cette nouvelle architecture constitue notre première contribution dans ces travaux.

Notre seconde contribution repose sur la mise en place d'une méthodologie pour sélectionner les composants logiciels permettant le fonctionnement de l'architecture. Effectivement, chaque couche logicielle peut impacter les performances liées à l'exécution de certaines instructions. Aussi, l'ajout d'une couche associée à un hyperviseur ne doit pas dégrader significativement les performances globales. Ce point est d'autant plus important que, dans le domaine aéronautique, les exigences liées aux contraintes temporelles sont strictes. L'analyse, à partir de cette méthodologie, a porté sur différents hyperviseurs. Celle-ci a permis d'identifier les composants de l'architecture permettant la mise en place d'une maquette.

Nous avons enfin validé notre démarche par la réalisation d'une maquette expérimentale. Cette maquette a été développé de manière à représenter au mieux l'architecture matérielle des équipements mobiles ainsi que les différentes couches logicielles utilisées.

Perspectives

Nous avons décrit, au chapitre III, un banc de test destiné à évaluer des hyperviseurs. Notre étude s'est portée sur deux solutions disponibles qui nous ont permis de valider notre démarche. Cependant, malgré de nombreux reports ou refus auxquels nous avons été confrontés, il serait utile d'insister auprès des industriels du secteur afin d'évaluer différentes solutions, notamment PolyXene qui présente l'avantage d'être certifié selon les Critères Communs. La méthodologie permettant d'analyser des surcharges liés aux couches peut être améliorée, par exemple, en augmentant le nombre des configurations. En effet, dans notre méthodologie, nous débutons nos mesures sur une configuration minimale permettant uniquement un accès matériel. Ensuite, nous ajoutons des couches logicielles de manière incrémentale jusqu'à parvenir à la configuration souhaitée. L'ajout d'une ou plusieurs configurations intermédiaires permettrait une analyse plus fine des différentes couches logicielles et ainsi identifier la ou les couches responsables de la principale surcharge observée, bien que celle-ci soit infime.

L'architecture que nous proposons prend pour cas d'étude un scénario de maintenance où des applications de niveaux de criticité différents sont amenées à communiquer. De nouveaux cas d'étude doivent être envisagés afin de valider et consolider la pertinence de notre architecture.

Ces travaux se sont focalisés sur la détection à l'exécution. Aussi, le démarrage de la machine n'est pas à négliger. En effet, si la machine est corrompue avant que l'application y soit installée, nous pouvons supposer que l'attaquant est susceptible de la contrôler et donc de modifier le modèle d'exécution ou plus généralement les mécanismes de comparaison afin de les mettre en défaut afin qu'aucune tentative d'attaque soit signalée. Dans l'architecture proposée, nous supposons que la machine est sûre lors de son démarrage. Afin de renforcer cette hypothèse, la création d'une chaîne de confiance assurant l'intégrité des différentes couches logicielles jusqu'à l'exécution de l'hyperviseur serait une piste à approfondir. L'utilisation d'une puce TPM (*Trusted Platform Module*) doit être prise en compte pour réaliser une telle chaîne de confiance.

Chapitre VI

Résumé

Traditionnellement, dans le domaine avionique les logiciels utilisés à bord de l'avion sont totalement séparés des logiciels utilisés au dehors afin d'éviter toutes interaction qui pourrait corrompre les systèmes critiques à bord de l'avion. Cependant, les nouvelles générations d'avions exigent plus d'interactions avec le monde ouvert avec pour objectif de proposer des services étendu, générant ainsi un flux d'information potentiellement dangereux. Dans une précédente étude, nous avons proposé l'utilisation de la virtualisation pour assurer la sûreté de fonctionnement d'applications critiques assurant des communications bidirectionnelles entre systèmes critiques et systèmes non sûr. Dans cette thèse nous proposons deux contributions.

La première contribution propose une méthode de comparaison d'hyperviseur. Nous avons développé un banc de test permettant de mesurer les performances d'un système virtualisé. Dans cette étude, différentes configurations ont été expérimentées, d'un système sans OS à une architecture complète avec un hyperviseur et un OS s'exécutant dans une machine virtuelle. Plusieurs tests (processeur, mémoire et réseaux) ont été mesurés et collectés sur différents hyperviseurs.

La seconde contribution met l'accent sur l'amélioration d'une architecture de sécurité existante. Un mécanisme de comparaison basé sur l'analyse des traces d'exécution est utilisé pour détecter les anomalies entre instances d'application exécutées sur diverse machines virtuelles. Nous proposons de renforcer le mécanisme de comparaison à l'exécution par l'utilisation d'un modèle d'exécution issu d'une analyse statique du bytecode Java.

Afin de valider notre approche, nous avons développé un prototype basé sur un cas d'étude identifié avec Airbus qui porte sur l'utilisation d'un ordinateur portable dédié à la maintenance.

Chapitre VII

Abstract

Traditionally, in avionics, on-board aircraft software used to be totally separated from open-world software in order to avoid any interaction that could corrupt critical on-board systems. However, new aircraft generations require more interaction with off-board systems to provide extended services, which makes these information flows potentially dangerous. In a previous work, we have proposed the use of virtualization to ensure dependability of critical applications despite bidirectional communication between critical on-board systems and untrusted off-board systems. In this thesis, we propose two contributions.

The first contribution concerns the establishment of a benchmark of hypervisors. We have developed a test bed to assess the performance impact induced by the use of virtualization. In this work, various configurations have been experimented ranging from a basic machine without an OS up to the complete architecture featuring a hypervisor and an OS running in a virtual machine. Several tests (computation, memory, and network) are carried out, and timing measures are collected on different hypervisors.

The second contribution focuses on the improvement of an existing security architecture. A comparison mechanism based on the analysis of execution traces is used to detect discrepancies between replicas supported by diverse virtual machines. We propose to strengthen the comparison mechanism at runtime by the use of an execution model, derived from a static analysis of the java bytecode.

To validate our approach, we have developed a prototype building on a case study identified with Airbus on a laptop dedicated to aircraft maintenance.

Bibliographie

[ADR Florida 2012] ADR Florida. *Advanced Data Research Florida.* `http://www.adrsoft.com/`, 2012.

[Aho *et al.* 2006] Alfred Aho, Monica Lam, Ravi Sethi, & Jeffrey Ullman. Compilers : Principles, techniques, and tools (2nd edition). Addison Wesley, 2006.

[APPLE 2011] *Contrat de licence de logiciel pour Mac OS X EA0730*, APPLE, 2011.

[ARINC 653 1997] Aeronautical Radio Incorporated (ARINC). *Avionics Application Standard Software Interface*, ARINC 653, 1997.

[ARINC 811 2005] Aeronautical Radio Incorporated (ARINC). *Commercial Aircraft Information Security Concepts of Operation and process Framework*, ARINC 811, 2005.

[Avizienis *et al.* 2004] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, & Carl Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Trans. Dependable Secur. Comput., vol. 1, no. 1, pages 11–33, janvier 2004.

[Avizienis 1985] Algirdas Avizienis. *The N-Version Approach to Fault-Tolerant Software.* IEEE Trans. Softw. Eng., vol. 11, no. 12, pages 1491–1501, décembre 1985.

[Ayache *et al.* 1979] Jean-Michel Ayache, Pierre Azéma, & Michel Diaz. *Observer : A concept for on-line detection of control errors in concurrent systems.* In The Ninth Fault-Tolerant Computing Symposium (FTCS-9), pages 79–86, 1979.

[Barham *et al.* 2003] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, & Andrew Warfield. *Xen and the art of virtualization.* In ACM SIGOPS Operating Systems Review, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. ACM ID : 945462.

[Bhukya *et al.* 2010] Devi Prasad Bhukya, Sirandas Ramachandram, & A.L. Reeta Sony. *Evaluating performance of sequential programs in virtual machine environments using*

design of experiment. In IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)., 2010.

[Blanquart *et al.* 2012] Jean-Paul Blanquart, Pierre Bieber, Gilles Descargues, Eric Hazane, Mathias Julien, & Léonardon Laurent. *Similarities and dissimilarities between safety levels and security levels*. In Embedded Real Time Software and Systems (ERTS2), février 2012.

[Bruneton *et al.* 2002] Eric Bruneton, Romain Lenglet, & Thierry Coupaye. *ASM : A code manipulation tool to implement adaptable systems*. In Adaptable and extensible component systems, 2002.

[Cabanes & Petit 2010] Reynold Cabanes & Jean-Luc Petit. *Observatoire des retards du transport aérien : Billan annuel 2010*. `http://www.developpement-durable.gouv.fr/spip.php?page=article&id_article=10339`, DGAC, 2010.

[CC 3.1 2009a] Members of the Common Criteria Recognition Arrangement (CCRA). *Common Criteria for Information Technology Security Evaluation Part 1 : Introduction and general model*, CC 3.1, 2009.

[CC 3.1 2009b] Members of the Common Criteria Recognition Arrangement (CCRA). *Common Criteria for Information Technology Security Evaluation Part 3 : Security assurance components*, CC 3.1, 2009.

[Cifuentes & Gough 1995] Cristina Cifuentes & K. John Gough. *Decompilation of binary programs*. Software : Practice and Experience, vol. 25, no. 7, pages 811–829, juillet 1995.

[CTCPEC 1993] Communications Security Establishment of Canada. CSEC. *The Canadian Trusted Computer Product Evaluation Criteria. Version 3.0e*, CTCPEC, 1993.

[Deswarte & Powell 2006] Yves Deswarte & David Powell. *Internet Security : An Intrusion-Tolerance Approach*. Proceedings of the IEEE, vol. 92, no. 2, pages 432–441, février 2006.

[Deswarte *et al.* 1991] Yves Deswarte, Laurent Blain, & Jean-Charles Fabre. *Intrusion tolerance in distributed computing systems*. In Proceedings of the IEEE Symposium on Research in Security and Privacy, pages 110–121, mai 1991.

[DO 178C 2011] European Organisation for Civil Aviation Equipment (EUROCAE). *Software considerations in airbone systems and equipement certification*, DO 178C, 2011.

[ED 202 2010] European Organisation for Civil Aviation Equipment (EUROCAE). *Airworthiness Security Process Specification*, ED 202, 2010.

[Eurocontrol and FAA 2007] Eurocontrol and FAA. *Communications Operating Concept and Requirements for the Future Radio System*. Rapport technique, 2007.

[FAA 2003a] FAA. *Airworthiness, and Operational Approval of Electronic Flight Bag Computing Devices*, Lignes directrice pour certification AC 120-76A, 2003.

[FAA 2003b] FAA. Faa order 8900.1 change 47 "aircraft equipement and operational authorizations", volume 4. 2003.

[Fabre *et al.* 1996] Jean-Charles Fabre, Yves Deswarte, & Laurent Blain. *Tolérance aux fautes et sécurité par fragmentation-redondance-dissémination.* Technique et Science Informatiques (TSI), vol. 15, pages 405–427, 1996.

[FCITS 1991] National Institute of Standards and Technology and National Security Agency. NIST and NSA. *Federal Criteria for Information Technology Security*, FCITS, 1991.

[Herndon *et al.* 2006] Bruce Herndon, Paula Smith, Eric Zamost, & Jennifer Anderson. *VMmark : A Scalable Benchmark for Virtualized Systems.* 2006.

[HLGAR 2011] High Level Group on Aviation and Aeronautics Research. HLGAR. *Flightpath 2050*, Étude prospective, 2011.

[ISO 27005 2008] International Organization for Standardization (ISO). *Gestion des risque liés à la sécurité de l'information*, ISO 27005, 2008.

[ITSEC 1991] Commission des communautés Européenne. CCE. *Critères d'évaluation de la sécurité des Systèmes informatiques*, ITSEC, 1991.

[JCSEC 1992] Ministry of International Trade and Industry. MITI. *The Japanese Computer Security Evaluation Criteria. Draft V1.0*, JCSEC, 1992.

[Jin *et al.* 1999] *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance*, Haoqiang Jin, Michael Frumkin, & Jerry Yan. `http://citeseerx.ist.psu.edu/viewdoc/summary`, 1999.

[John 1999] Rushby John. *Partitioning in Avionics Architectures : Requirements, Mechanisms, and Assurance.* Rapport technique, 1999.

[Kaiser & Wagner 2007] R. Kaiser & S. Wagner. *The PikeOS Concept - Hystory and Design.* Rapport technique, 2007.

[Kiczales *et al.* 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, & John Irwin. *Aspect-oriented programming.* In ECOOP. Springer-Verlag, 1997.

[Kiczales *et al.* 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, & William G. Griswold. *An Overview of AspectJ.* pages 327–353. Springer-Verlag, 2001.

[Kontagora & Gonzalez-Velez 2010] Maryam Kontagora & Horacio Gonzalez-Velez. *Benchmarking a MapReduce Environment on a Full Virtualisation Platform.* In International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pages 433–438, 2010.

[Kuleshov 2007] E. Kuleshov. *Using the ASM framework to implement common Java bytecode transformation patterns.* In Proceedings of the 6th International Conference on Aspect-Oriented Software Development., Vancouver, BC, Canada, March 2007.

[Laarouchi 2009] Youssef Laarouchi. *Sécurités (immunité et innocuité) des architectures ouvertes à niveaux de criticité multiples : application en avionique.* Thèse de doctorat, Institut National Science Appliquées de Toulouse, novembre 2009.

[Laprie *et al.* 1990] Jean-Claude Laprie, Jean Arlat, Christian Béounes, & Karama Kanoun. *Definition and analysis of hardware- and software-fault-tolerant architectures.* Computer, vol. 23, no. 7, pages 39 –51, juillet 1990.

[Laprie *et al.* 1996] Jean-Claude Laprie, Jean Arlat, Yves Deswarte, David Powell, Karama Kanoun, Mohamed Kaâniche, & Yves Crouzet. Guide de la sûreté de fonctionnement. Cépadues, 1996.

[Lastera *et al.* 2011] Maxime Lastera, Eric Alata, Jean Arlat, Yves Deswarte, Bertrand Leconte, David Powell, & Cristina Simache. *Characterization of Hypervisor for Security-Enhanced Avionics Applications.* In SAE Aerospace Electronics and Avionics Systems Congress, octobre 2011.

[Lastera *et al.* 2012] Maxime Lastera, Eric Alata, Jean Arlat, Yves Deswarte, Bertrand Leconte, David Powell, & Cristina Simache. *Secure architecture for information systems in avionics.* In Embedded Real Time Software and Systems (ERTS2), 2012.

[Lee & Anderson 1990] Pete. A. Lee & Tom. Anderson. Fault tolerance : Principles and practice. Springer-Verlag, Secaucus, NJ, USA, 2nd edition, 1990.

[Lindholm & Yellin 1999] Tim Lindholm & Frank Yellin. Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[Lorczak *et al.* 1989] Paul.R Lorczak, Alper.K Caglayan, & Dave.E Eckhardt. *A theoretical investigation of generalized voters for redundant systems.* In Digest of Papers., Nineteenth International Symposium on Fault-Tolerant Computing FTCS-19, pages 444 –451, juin 1989.

[MAFTIA 2003] Adelsbach André, Alessandri Dominique, Cachin Christian, Creese Sadie, Deswarte Yves, Kursaw Klause, Laprie Jean-Claude, Powell David, Randell Brian, Riordan James, Ryan Peter, Simmonds

William, Stroud Robert, Veríssimo Paulo, Waidner Michael, & Wesp Andreas. *Malicious and Accidental Fault Tolerance for Internet Applications : Conceptual Model and Architecture*, MAFTIA, janvier 2003.

[McDougall & Anderson 2010] Richard McDougall & Jennifer Anderson. *Virtualization performance : perspectives and challenges ahead.* ACM SIGOPS Operating Systems Review, vol. 44, pages 40–56, décembre 2010. ACM ID : 1899933.

[McVoy & Staelin 1996] Larry McVoy & Carl Staelin. *lmbench : portable tools for performance analysis.* In Proceedings of the Annual Technical Conference USENIX, Berkeley, CA, USA, 1996. USENIX Association. ACM ID : 1268322.

[Morgan 1991] M.J. Morgan. *Integrated modular avionics for next-generation commercial airplanes.* In Proceedings of the IEEE National Aerospace and Electronics Conference. NAECON, volume 1, pages 43–49, mai 1991.

[Namjoo 1982] Mohsen Namjoo. Cerberus-16 : an architecture for a general purpose watchdog processor. Center for Reliable Computing, Computer Systems Laboratory, Stanford University, 1982.

[Peterson & Weldon 1972] William Wesley Peterson & E. J. Weldon. Error-Correcting codes. MIT Press, mars 1972.

[PhysTech 2005] PhysTech. *Ubench.* `http://phystech.com/download/ubench.html`, 2005.

[PIPAME 2009] Pôle interministériel de prospective et d'anticipation des mutations économiques. PIPAME. *La chaîne de valeur dans l'industrie aéronautique*, Étude prospective. `http://www.industrie.gouv.fr/p3e/etudes-prospectives`, septembre 2009.

[Poledna 1994] Stefan Poledna. *Replica determinism in distributed real-time systems : a brief survey.* Real-Time Syst., vol. 6, no. 3, pages 289–316, mai 1994.

[Rabéjac 1995] Christophe Rabéjac. *Auto-surveillance logicielle pour applications critiques.* Thèse de doctorat, Institut National Polytechnique de Toulouse, 1995.

[Randell 1975] Brian Randell. *System structure for software fault tolerance.* In Proceedings of the international conference on Reliable software, pages 437–449, New York, NY, USA, 1975. ACM.

[Rutkowska 2010] Joanna Rutkowska. Qubes OS architecture, version 0.3. 2010.

[Seungkwon & Youngil 2007] Cho Seungkwon & Kim Youngil. *Linux BYTEmark Benchmarks : A Performance Comparison of Embedded Mobile Processors.* In The 9th International Conference on Advanced Communication Technology, volume 1, pages 125–127, février 2007.

[SFP 2001a] SFP. *Webpage of the NEVADA project.* Rapport technique, 2001.

[SFP 2001b] SFP. *Webpage of the PAMELA project*. Rapport technique, 2001.

[SFP 2001c] SFP. *Webpage of the VICTORIA project*. Rapport technique, 2001.

[SFP 2011] SFP. *Website of the SCARLETT project*. Rapport technique, 2011.

[Sheng 1999] Liang Sheng. Java native interface : Programmer's guide and reference. Addison-Wesley Longman Publishing CO, Boston, MA,USA, 1st edition, 1999.

[Skaves 2011] Peter Skaves. *Electronic flight bag (EFB) policy and guidance*. In Digital Avionics Systems Conference, octobre 2011.

[Smith & Nair 2005] James E. Smith & Ravi Nair. *The Architecture of Virtual Machines*. Computer, vol. 38, no. 5, pages 32–38, mai 2005.

[TASF 2007] The Apache Software Foundation. TASF. *Hadoop*. `http://hadoop.apache.org/`, 2007.

[TCSEC 1985] DOD US Department of defense. DoD. *Trusted Computer Security Evaluation Criteria, 5200.28-STD*, TCSEC, 1985.

[TLSDRG 2002] The Last Stage of Delirium Research Group. TLSDRG. *Java and Java Virtual Machine. Security vulnerabilities and their exploitation techniques*. Black Hat Briefings, 2002.

[Totel 1998] Éric Totel. *Politique d'intégrité multiniveau pour la protection en ligne de tâches critiques*. Thèse de doctorat, Institut National Polytechnique de Toulouse, décembre 1998.

[Traverse *et al.* 2004] Pascal Traverse, Isabelle Lacaze, & Jean Souyris. *Airbus Fly-By-Wire : A Total Approach To Dependability*. In Building the Information Society, volume 156, pages 191–212. Springer Boston, 2004.

[Umeno *et al.* 2006] Hidenori Umeno, M.L.C. Parayno, K Teramoto, M Kawano, H Inamasu, S Enoki, Masato Kiyama, Tomonori. Aoyama, & Takafumi Fukunaga. *Performance Evaluation on Server Consolidation Using Virtual Machines*. In SICE-ICASE, International Joint Conference, 2006.

[Walters 1999] Brian Walters. *VMware Virtual Platform*. Linux Journal, vol. 1999, no. 63, juillet 1999. ACM ID : 327912.

[Xu *et al.* 2008] Xianghua Xu, Feng Zhou, Jian Wan, & Yucheng Jiang. *Quantifying Performance Properties of Virtual Machine*. In International Symposium on Information Science and Engineering, ISISE '08., volume 1, pages 24–28, 2008.