Analyse de vulnérabilités de systèmes avioniques embarqués : classification et expérimentation

Anthony Dessiatnikoff

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *17/07/2014* **par :**

ANTHONY DESSIATNIKOFF

Analyse de vulnérabilités de systèmes avioniques embarqués : classification et expérimentation

---

JURY

JEAN-LOUIS LANET	Université de Limoges	Rapporteur
OLIVIER FESTOR	Télécom Nancy	Rapporteur
BERTRAND LECONTE	Ingénieur Airbus	Examinateur
COLAS LE GUERNIC	Ingénieur DGA	Examinateur
VINCENT NICOMETTE	INSA de Toulouse	Examinateur
ERIC ALATA	INSA de Toulouse	Directeur de thèse

École doctorale et spécialité :

MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :

Laboratoire d'Analyse et d'Architecture des Systèmes (UPR 8001)

Directeur de Thèse :

Eric ALATA

Rapporteurs :

Jean-Louis LANET et *Olivier FESTOR*

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 Raja Chatila, Jean-Louis Sanchez et Jean Arlat qui ont assuré la direction du LAAS depuis mon entrée.

Je remercie également Karama Kanoun et Mohamed Kaâniche, responsables successifs de l'équipe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser mes travaux au sein du groupe.

Je remercie la Direction Générale de l'Armement (DGA) et le CNRS pour m'avoir financé pendant 3 années puis l'Agence Nationale de la Recherche avec le projet SOBAS pendant les derniers mois.

Je tiens à exprimer ma profonde reconnaissance et un grand respect à Yves Deswarte, Directeur de Recherche au CNRS, pour m'avoir accueilli en stage au LAAS en 2009 et m'avoir fait confiance pour la réalisation de ces travaux de thèse en 2010 et pour avoir été mon directeur de thèse pendant 3 années. Sa disparition le 27 janvier 2014 a été une profonde douleur.

J'exprime ma gratitude et remercie Éric Alata et Vincent Nicomette pour m'avoir encadré avec Yves Deswarte pendant ma thèse. Ils étaient toujours disponibles pour discuter et corriger mes erreurs, malgré leurs emplois du temps très chargés.

J'adresse également mes sincères remerciements aux membres du jury qui ont accepté de juger mon travail. Je leur suis très reconnaissant pour l'intérêt qu'ils ont porté à mes travaux :

- Jean-Louis Lanet, Professeur des Universités, laboratoire XLIM de Limoges
- Olivier Festor, Professeur des Universités, Télécom Nancy
- Vincent Nicomette, Professeur des Universités, INSA de Toulouse
- Bertrand Leconte, Ingénieur de recherche, Airbus, Toulouse
- Colas Le Guernic, Ingénieur de recherche, DGA, Rennes

Je remercie particulièrement Messieurs Jean-Louis Lanet et Olivier Festor qui ont accepté la charge d'être rapporteurs.

Je remercie les nombreuses personnes ayant participé, depuis le début ou en partie, au projet SOBAS de l'ANR : Bertrand Leconte (Airbus), Benoît Triquet (Airbus), Alain Combes (Airbus), Cristina Simache (Altran), Julien Iguchi-Cartigny (XLIM), Jean-Paul Blanquart (ASTRIUM), Stéphane Duverger (EADS), les membres de l'ANSSI ainsi que les autres personnes que j'ai pû oublier.

Une mention particulière est donnée à Benoît Triquet (Airbus) et Fabrice Thorignac (Atos Origin) pour m'avoir accueilli dans les locaux d'Airbus plusieurs jours pendant ces 3 années et qui m'ont énormément aidé pour la compréhension du code noyau, des applications avioniques et du P4080. Je remercie également Stéphane Duverger pour son enthousiasme permanent et sa grande expertise sur les systèmes d'exploitation. De plus, je remercie Cristina Simache ainsi que les autres personnes qui ont relu et corrigé le manuscrit.

Je remercie également tous les doctorants, docteurs, post-doctorants et stagiaires de l'équipe TSF que j'ai pû rencontrer, je ne peux pas tous les citer mais je vais essayer : Miruna, Carla, Fernand, Miguel, Camille, Jesus, Pierre, Thibaut, Amira, Quynh Anh, Moussa, Hélène, Ludovic, Olivier, Benoît, Ivan, Yann, Joris, Roberto, Mathilde, Kalou, Julien, Guthemberg, Fanny, Robert, Hongli, Roxana, Irina, Kossi, Jimmy, Amina, Rim, Damien, Éric, Géraldine, etc. dont je me suis lié d'amitié avec une grande partie. Je remercie tous les permanents TSF car ils contribuent pour beaucoup à la bonne entente générale dans l'équipe TSF. J'ai eu la chance de rencontrer au LAAS beaucoup de monde très intéressant, j'ai une pensée pour Jean-Claude Laprie (qui était un grand monsieur, dans tous les sens du terme).

De plus, je remercie Sonia De Sousa pour sa gestion parfaite du secrétariat du thème Informatique Critique du laboratoire, elle a toujours été disponible et très efficace pour m'aider dans les différentes tâches administratives.

Je pense également à mes autres amis que je vois trop peu à cause de la distance et qui m'ont soutenu : Richard, Antoine et Mellie, Damien et Caroline, Steven et Lucie, et les autres.

Je tiens à remercier particulièrement ma famille qui m'a toujours soutenu depuis le début, même s'ils n'ont pas toujours compris ce que je faisais exactement, ils ont le mérite de m'avoir encouragé dans les moments difficiles.

« La vraie faute est celle qu'on ne corrige pas. »

Confucius (551 – 479 avant J.C.)

Sommaire

Table des figures

Introduction générale

Contexte et problématique

La sécurité des systèmes embarqués avioniques, au sens *safety* (ou sécurité-innocuité) est depuis toujours une préoccupation majeure des concepteurs et des exploitants de ces systèmes et plus largement du grand public. Historiquement, les sources de défaillance de ces systèmes étaient essentiellement liées à l'occurrence de fautes accidentelles, affectant les composants matériels ou logiciels, ou bien à des interactions humaines inappropriées. Cependant, le rôle de plus en plus prépondérant joué par les systèmes informatiques au sein des systèmes avioniques et également l'impact de leurs communications avec d'autres systèmes de l'environnement avionique posent de nouvelles problématiques relatives à la sécurité (au sens *security* ou sécurité-immunité) et aux risques d'attaques informatiques qui viennent s'ajouter aux contraintes de *safety* (ou sécurité-innocuité).

Cette importance de la notion de sécurité-immunité est liée à l'évolution des systèmes d'information qui vise à réduire les coûts de production et d'opération. Cette évolution repose principalement sur trois approches. Tout d'abord, l'industrie aéronautique propose des systèmes d'information de plus en plus ouverts aux passagers et aux compagnies aériennes. Cette approche entraîne un élargissement de la surface d'attaque. Ensuite, 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. Pour finir, le partage des ressources est de plus en plus employé pour héberger, au sein d'un même système, différentes applications ayant potentiellement différentes exigences de sécurité.

Pour améliorer le niveau de sécurité des systèmes embarqués avioniques critiques, des techniques rigoureuses de développement, s'appuyant de plus en plus sur des méthodes formelles, sont utilisées pour accompagner les phases de spécification, de conception, de vérification et de certification, dans le but de contrôler si les propriétés de sécurité sont assurées. Cependant ces méthodes ne peuvent pas prendre en compte toutes les malveillances, notamment celles qui visent à exploiter certaines vulnérabilités invisibles par les méthodes formelles. Ces vulnérabilités exploitables se trouvent à l'interface entre

logiciel et matériel, en particulier dans l'implémentation par le matériel de fonctions exploitées par le logiciel (gestion de l'adressage, gestion des interruptions, mécanismes de tolérance aux fautes, etc.) mais aussi sur des fonctions annexes de gestion du matériel (contrôle de fréquence d'horloge, de la consommation, de la température, de l'alimentation, etc.). Ces aspects ne sont généralement pas considérés par des modèles formels. En effet, il est difficile pour ce type de modèles de prendre en compte à la fois des propriétés de sécurité de haut niveau (par exemple l'intégrité des données critiques) et une représentation fine d'une implémentation matérielle. L'approche formelle doit donc être complétée par des analyses de vulnérabilités. Traditionnellement, ces analyses ne sont réalisées que lorsque le système est complètement intégré, donc très tard dans le cycle de développement, ce qui ne permet que de corriger ponctuellement des fautes de conception ou d'implémentation.

L'approche développée dans cette thèse vise donc à compléter les méthodes formelles de conception et de développement par une analyse de vulnérabilités de façon à identifier au plus tôt des classes génériques de vulnérabilités, soit pour modifier la conception, soit pour développer des contre-mesures génériques, capables d'empêcher l'exploitation des vulnérabilités identifiées, mais aussi d'autres vulnérabilités similaires non encore connues.

Ces travaux ont été effectuées dans le cadre du projet ANR SOBAS (*Securing On-Board Aerospace Systems*).

Présentation de nos travaux

Nous débutons ce manuscrit par un chapitre dédié à la présentation des architectures des systèmes avioniques ainsi que leur évolution, étroitement liée aux avancées significatives du domaine du logiciel. Ensuite, nous présentons les approches employées pour améliorer la sécurité-immunité dans ces systèmes et nous nous focalisons sur l'analyse de vulnérabilités, qui est l'objet de ce manuscrit. Nous concluons ce chapitre par la proposition d'une démarche en trois étapes pour l'application de l'analyse de vulnérabilités. Chacune de ces étapes fait ensuite l'objet d'un chapitre.

La première étape, présentée dans le second chapitre, est dédiée à la caractérisation des attaques des systèmes embarqués. Les systèmes étant composés de différents composants matériels et logiciels, ces composants sont passés en revue pour identifier les attaques qui peuvent les concerner. Les composants passés en revue sont les composants matériels, les composants logiciels du système d'exploitation et les mécanismes de tolérance aux fautes. L'objectif de ce chapitre est de proposer une approche pour recenser l'ensemble des risques possibles, qui doivent faire l'objet d'une étude plus approfondie.

La seconde étape se focalise sur la mise en œuvre des attaques sur une plateforme d'expérimentation. Ce travail est présenté dans le troisième chapitre. Tout d'abord, nous détaillons les différents éléments constituant notre plateforme d'expérimentation. Ensuite, quatre classes d'attaques correspondant aux attaques les plus importantes sont présentées, et les résultats sont discutés.

Le quatrième chapitre présente la dernière étape de notre approche. Cette étape passe à nouveau en revue les classes d'attaques identifiées lors de la caractérisation, et identifie,

pour chacune de ces classes d'attaques, un ensemble de contre-mesures adaptées. Ce chapitre est structuré en trois parties. Les contre-mesures spécifiques aux vulnérabilités identifiées dans le troisième chapitre sont dans un premier temps discutées. Ensuite, ces contre-mesures sont généralisées pour être adaptées à tout système embarqué. Pour finir, une architecture matérielle et logicielle, permettant le partage d'un périphérique entre plusieurs applications, est présentée et illustrée avec une carte réseau.

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

Chapitre 1

État de l'art de la sécurité des systèmes avioniques

Introduction

Selon le US-CERT[1], le nombre d'attaques informatiques est en forte augmentation depuis 2006 [usc12]. Cette augmentation peut s'expliquer par deux raisons : la première est la complexité des systèmes actuels qui est de plus en plus élevée et donc plus difficile à gérer [Lab12] ; la deuxième raison est la simplicité d'utilisation d'outils automatiques d'attaque, maintenant accessibles aux débutants. Ces outils peuvent par exemple être utilisés pour réaliser des attaques de type *Distributed-Denial-of-Service*[2]. A titre d'exemple, le groupe de pirates Anonymous a lancé une série d'attaques en janvier 2012 après la fermeture du site de partage Megaupload, ce qui a rendu indisponible de nombreux sites Internet [Can12]. Un autre exemple d'attaque de plus grande envergure est celle d'août 2013 qui a touché la Chine en bloquant les accès à une grande partie des sites Internet Chinois [01N13]. Avec la même facilité, les attaques contre des applications Web sont aussi possibles avec des outils tels que WebScarab [OWA13]. Ces outils en libre circulation sont performants et permettent de lancer des attaques contre n'importe quel système distant. Il s'agit de programmes malveillants qui peuvent endommager de façon catastrophique les systèmes ciblés : accès, modification de données confidentielles ; espionnage du système (*keylogger*) ; interruption de services (*Denial-of-Service*) ; exécution de virus ou *rootkit* ; attaque par force brute pour obtenir des mots de passe ; etc.

Les systèmes embarqués avioniques peuvent également être ciblés par ces attaques. Il est donc important de les protéger efficacement contre ces malveillances. Une attaque informatique visant un système avionique peut avoir de graves conséquences comme des pertes matérielles voire des pertes humaines. Afin d'éviter l'occurrence de défaillances dans les systèmes avioniques, des mécanismes de tolérance aux fautes sont utilisés (redondance, votes majoritaires, etc.). Cependant, ces mécanismes sont efficaces pour des défaillances d'origine accidentelle mais ils ne sont pas forcément efficaces face à des at-

[1] *United States Computer Emergency Readiness Team.*

[2] Technique distribuée pour rendre hors service un serveur par inondation de requêtes.

taques (autrement dit, face à des fautes d'origine intentionnelle). Il est donc nécessaire de considérer aussi d'autres moyens de protection, spécifiquement développés pour faire face aux fautes intentionnelles que sont les attaques informatiques. Et même si de tels moyens sont déjà utilisés dans les avions aujourd'hui, ils ne sont pas dans la même état de maturité que les mécanismes de tolérance aux fautes accidentelles. De plus, les architectures des systèmes embarqués avioniques évoluant, les menaces considérées évoluant également, il est donc fondamental d'améliorer ces mécanismes et de les adapter aux nouvelles architectures.

Dans ce chapitre, nous décrivons une architecture utilisée pour la conception de système avionique. Ensuite, nous présentons les normes avioniques qui permettent de s'assurer que le code utilisé est conforme à sa spécification et ne contient pas de faute. Ensuite, nous détaillons comment ces systèmes sont protégés vis-à-vis des fautes accidentelles (sécurité-innocuité ou *safety*) et vis-à-vis des malveillances (sécurité-immunité ou *security*). Nous concluons ce chapitre en présentant une méthodologie d'analyse de vulnérabilités permettant d'identifier les vulnérabilités lors du développement afin d'intégrer au système les mécanismes de protection les plus adaptés, avant la finalisation du développement du système.

1.1 Systèmes 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 [pip09]).

1.1.1 Architecture fédérée

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 1.1. 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-à-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.

Un avantage majeur des architectures fédérées, 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 ar-

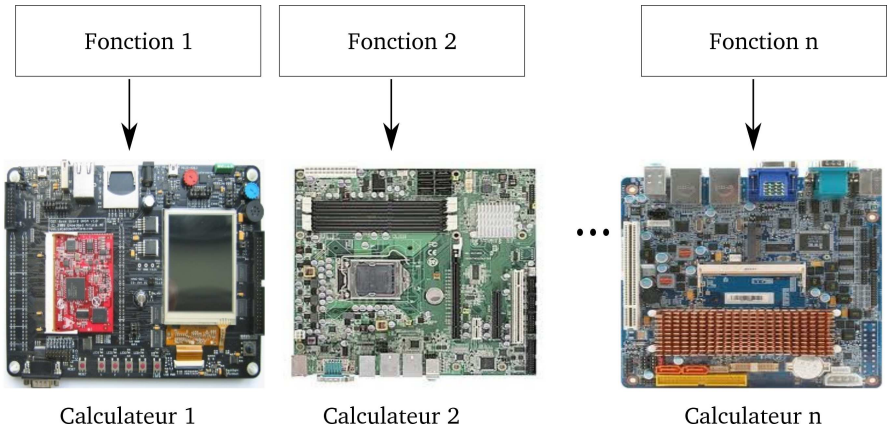

FIG. 1.1 – Architecture fédérée

chitectures 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 une 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.

1.1.2 Architecture IMA

Dans les années 1990, sous l'impulsion de la commission européenne, des projets européens de recherche tel que PAMELA, NEVADA et VICTORIA [PAM01, NEV01, VIC01], 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*) [ARI08]. Le principe de l'architecture IMA [Mor91] 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 :

- Réduction du poids grâce à un plus petit nombre de composants matériels et un câblage réduit, ce qui augmente l'efficacité énergétique ;
- Coût de maintenance réduit par l'utilisation de modules génériques ;
- Réduction des coûts de développement par l'utilisation de systèmes standardisés et de COTS (*Commercial-Off-The-Shelf*).

Cette organisation du système présente une plateforme unique (*cf.* Figure 1.2) pour plusieurs 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 [ARI08]. Par ce moyen, une ou plusieurs applications regroupées au sein d'un même module peuvent s'exécuter de manière sûre en parallèle (deux fonctions quelconques prévues pour s'exécuter dans un même module ne peuvent en aucun cas interagir l'une avec 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).

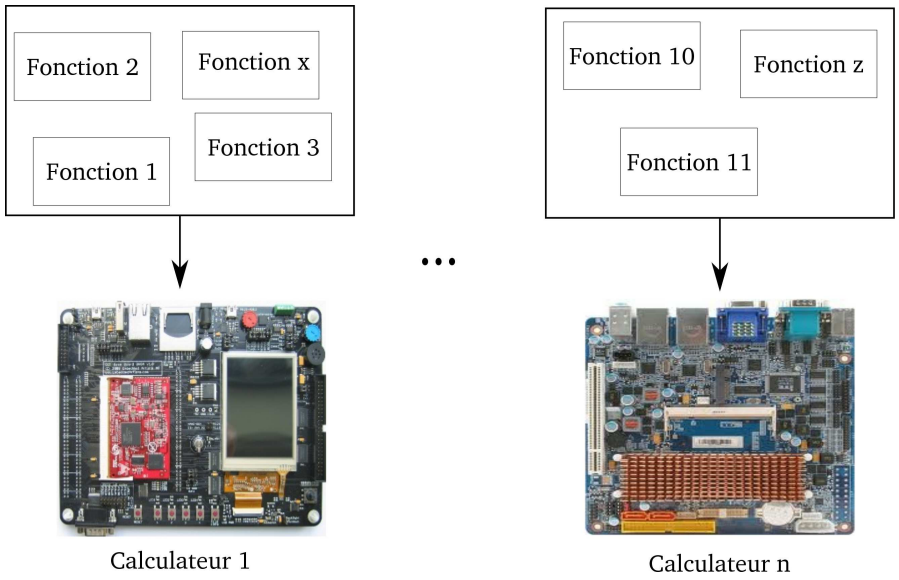

FIG. 1.2 – Architecture IMA

Ce partitionnement est classé en partitionnement spatial (par exemple, la gestion de la mémoire) et temporel (la gestion des ressources temporelles par le CPU) [Rus99]. 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 rafraîchissement de paramètres critiques). Avec ce partitionnement, des tranches de temps pour l'exécution des applications et l'exécution du noyau du système sont allouées statiquement. Afin de déterminer la durée des tranches de temps, des études d'estimation du pire cas du temps d'exécution (WCET[3]) sont réalisées de façon dynamique (sur le matériel ou sur simulateur) ou de façon statique (sans exécution) [Pua11]. Les méthodes d'estimation du WCET ne sont définies dans aucune norme avionique, il faut donc choisir la méthode appropriée en fonction de ses besoins. Il faut également prendre en compte l'utilisation de plusieurs cœurs dans les processeurs aujourd'hui car les multi-cœurs tendent à remplacer les mono-cœurs [Gen13, ZY09] ce qui change considérablement les calculs des durées d'exécution. La méthode dynamique d'évaluation du WCET s'appuie sur l'exécution de l'application sur le matériel ou sur simulateur et la mesure du temps d'exécution entre le début et la fin d'exécution. Le comportement de l'application peut dépendre des entrées/sorties. Il est donc nécessaire de générer ces informations. Une couverture

[3] *Worst Case Execution Time.*

exhaustive du domaine des entrées serait bien sûr souhaitable mais le risque d'explosion combinatoire est élevé. En général, il s'agit donc de trouver un compromis entre la couverture des entrées générées et la précision du WCET que l'on souhaite obtenir. Un outil peut être utilisé pour mesurer le WCET automatiquement. Par exemple, RapiTime[4] permet de créer des traces d'exécution et supporte la norme DO-178B (norme avionique sur laquelle nous reviendrons dans la suite de ce chapitre). La méthode statique d'évaluation du WCET calcule la durée d'exécution en analysant la structure du programme. Différentes techniques d'analyse existent : analyse des flux (*Flow analysis*), calcul avec arbre (*Tree-based computation*), énumeration des chemins, (IPET, *Implicit Path Enumeration Technique*), analyse de bas niveau (*Low-level analysis*). Mais en pratique, ces durées d'exécution seront forcément dépendantes du matériel et des exécutions passées. Elles doivent être calculées dans toute la chaîne d'exécution, c'est-à-dire qu'il faut être capable de connaître le chemin d'exécution exact ainsi que chaque type de données à traiter (entier, chaînes de caractères, etc.). De plus, les durées doivent être calculées pour tous les chemins d'exécution. Des outils sont également disponibles pour une méthode d'estimation statique, nous pouvons citer Bound-T[5], AbsInT[6] (notamment utilisé pour l'A380).

De manière générale, le partitionnement spatial et temporel 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 [ARI08], 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 1.2), 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 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 1.3 présente l'aspect en couche d'un module IMA. Le module est constitué

[4] `http://www.rapitasystems.com/products/rapitime`
[5] `http://www.bound-t.com/`
[6] `http://www.absint.com/ait/`

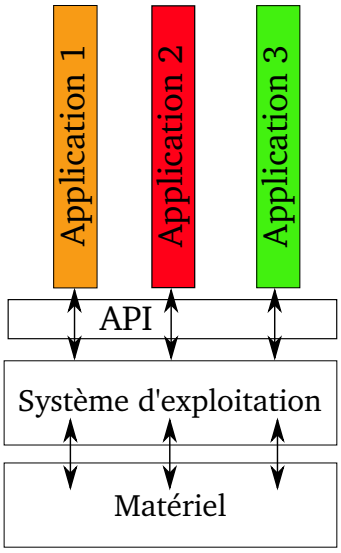

FIG. 1.3 – Exemple de Module IMA

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.* 1.2.2), 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 [KW07] 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 [SCA11] 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. Ces deux propriétés sont plus précisément définies dans le cadre de la sûreté de fonctionnement.

1.2 Sûreté de fonctionnement et le développement logiciel

Dans cette section, nous donnons des définitions de la sûreté de fonctionnement et de ses composantes pour comprendre quelle utilité elle peut avoir dans un contexte avionique. Nous détaillons également comment s'effectue le développement de logiciels critiques en utilisant des normes et techniques de la sûreté de fonctionnement. Les définitions de cette section proviennent de [L+96, ALRL04, A+06].

1.2.1 Quelques définitions de la sûreté de fonctionnement

La **sûreté de fonctionnement** d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Le 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 physique qui interagit avec le système considéré. La sûreté de fonctionnement peut être représentée par l'arbre suivant :

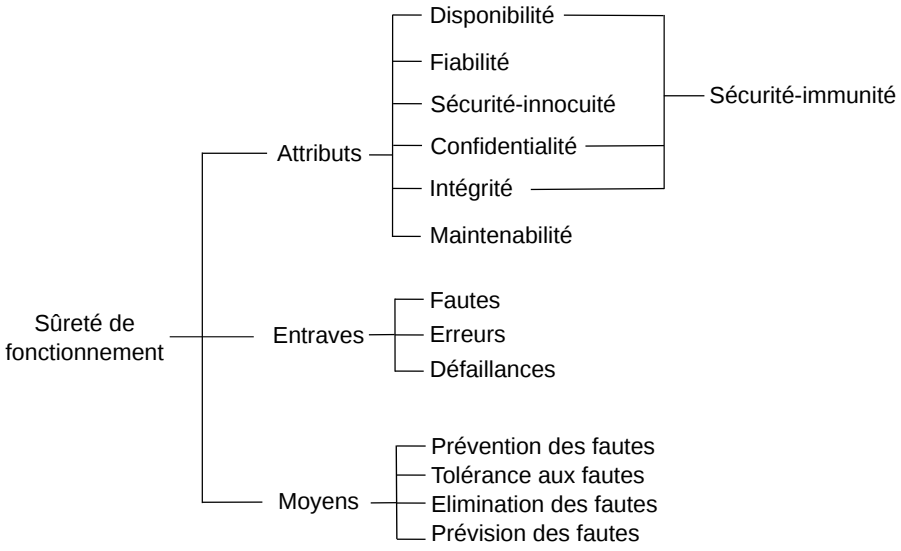

FIG. 1.4 – L'arbre de la sûreté de fonctionnement

Cet arbre permet de définir les termes importants de la sûreté de fonctionnement : les attributs, les entraves et les moyens.

1.2.1.1 Attributs

La sûreté de fonctionnement peut être perçue selon différentes propriétés appelées attributs :

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

Ces attributs sont à considérer suivant le système que l'on souhaite étudier. Cela permet également de différencier sans ambiguïté les termes de *safety* et *security*. Le terme *safety* correspond à la sécurité-innocuité, qui définit les propriétés selon lesquelles un système est dit "sûr" (sans défaillance catastrophique pouvant conduire à des pertes de vies humaines ou des conséquences économiques importantes). Le terme *security* correspond à la **sécurité-immunité**[7], qui définit les propriétés selon lesquelles un système est dit "sécurisé". Cet attribut est une combinaison de trois autres attributs : la confidentialité, l'intégrité et la disponibilité. La sécurité-immunité permet de prendre en compte les menaces informatiques tels que les virus, vers, bombes logiques, etc.

1.2.1.2 Entraves

Les entraves sont les circonstances indésirables mais non inattendues, causes ou résultats de la non sûreté de fonctionnement, car la confiance n'est plus assurée dans le système. Les entraves à la sûreté de fonctionnement sont les suivantes : **fautes**, **erreurs** et **défaillances**. La relation entre les entraves de la sûreté de fonctionnement peut être exprimée ainsi :

$$\ldots \rightarrow \text{Défaillance} \rightarrow \text{Faute} \rightarrow \text{Erreur} \rightarrow \text{Défaillance} \rightarrow \text{Faute} \rightarrow \ldots$$

Une faute est active (après avoir été dormante) lorsqu'elle produit une erreur. L'ensemble des fautes peuvent être réparties en huit catégories, selon : la phase de création (fautes de développement ou fautes opérationnelles), les frontières du système (internes ou externes), les causes phénoménologiques (naturelles ou humaines), les objectifs (fautes malicieuses ou non), la dimension (fautes matérielles ou fautes logicielles), les intentions (fautes délibérées ou non), la capacité (fautes accidentelles ou non) ou la persistance (fautes permanentes ou éphémères). Ces fautes peuvent être combinées pour former trois grandes classes de fautes : les fautes de développement, les fautes physiques et les fautes d'interaction. Les **fautes de développement** constituent toutes les fautes qui surviennent pendant le développement du logiciel. Les **fautes physiques** incluent les fautes qui affectent le matériel. Les **fautes d'interaction** incluent les fautes externes.

Une erreur est définie comme susceptible de provoquer une défaillance. Elle peut être latente ou détectée. Une erreur est latente tant qu'elle n'a pas été reconnue en tant

[7] Aussi appelée sécurité-confidentialité dans [L+96].

que telle ; elle est détectée par un algorithme ou un mécanisme de détection qui permet de la reconnaître comme telle. Une erreur peut disparaître avant d'être détectée. Par propagation, une erreur crée de nouvelles erreurs.

Une défaillance survient lorsque, par propagation, elle affecte le service délivré par le système, donc lorsqu'elle est perçue par le ou les utilisateurs. Une défaillance est définie comme un événement qui survient quand le service délivré dévie du service correct. Les défaillances ne se manifestent pas toujours de la même manière, ce qui signifie que l'on peut parler de modes de défaillances. On peut distinguer différentes dimensions pour caractériser les défaillances : le domaine de défaillance, la détectabilité de défaillances, la cohérence des défaillances, et les conséquences des défaillances. Le **domaine des défaillances** permet de distinguer les défaillances en valeur (la valeur du service délivré dévie de l'accomplissement de la fonction du système) et les défaillances temporelles (la temporisation du service dévie de l'accomplissement de la fonction du système). La **détectabilité des défaillances** permet de distinguer les défaillances signalées (le service délivré est détecté comme incorrect) et non signalées (la délivrance d'un service incorrect n'est pas détectée). La **cohérence des défaillances** permet de distinguer les défaillances cohérentes et les défaillances incohérentes ou byzantines (le service incorrect peut être perçu différemment suivant les cas). Les **conséquences des défaillances** permettent de distinguer les défaillances bénignes (les conséquences sont du même ordre de grandeur que le bénéfice procuré par le service délivré en l'absence de défaillance) des défaillances catastrophiques (les conséquences sont incommensurablement différentes du bénéfice procuré par le service délivré en l'absence de défaillance).

1.2.1.3 Moyens

Quatre grandes catégories sont utilisées comme moyens pour satisfaire les attributs de la sûreté de fonctionnement :

- **La prévention des fautes** permet d'empêcher l'occurrence ou de l'introduction d'une faute.
- **La tolérance aux fautes** permet de fournir un service qui remplit la fonction du système en dépit de fautes.
- **L'élimination des fautes** permet de réduire le nombre et la sévérité des fautes.
- **La prévision des fautes** permet d'estimer la présence, la création et les conséquences des fautes.

La volonté d'empêcher l'occurrence de fautes paraît évidente pour des développeurs et notamment pour le développement de logiciels et la conception de matériels sûrs. Plusieurs techniques peuvent être utilisées : méthodes formelles, utilisation de langages fortement typés, règles de conception rigoureuses, etc.

Un système tolérant aux fautes doit pouvoir remplir ses fonctions en dépit de fautes pouvant affecter ses composants, sa conception ou ses interactions avec des hommes ou d'autres systèmes. Pour mettre en œuvre la tolérance aux fautes, des techniques de **traitement d'erreurs** et de **traitement de fautes** sont utilisées. Parmi les différentes techniques de traitement d'erreurs, on peut citer la compensation, la reprise (*rollback*) ou la poursuite d'erreurs (*rollforward*). Parmi les différentes techniques de traitement de

fautes, on peut citer le diagnostic, l'isolation, la reconfiguration ou la réinitialisation.

L'élimination des fautes peut être faite pendant la phase de développement ou pendant la phase d'exécution. Pendant la phase de développement, l'élimination des fautes est réalisée en trois étapes : **la vérification**, **le diagnostic** et **la correction**. La vérification permet de vérifier que le système est exempt de fautes. Si une faute est détectée, le système de diagnostic a pour objectif d'en connaître la raison puis le système de correction permet de corriger cette faute. Pendant l'exécution, l'élimination des fautes consiste en de la maintenance corrective (après avoir détecté une faute) ou préventive (avant d'avoir détecté une faute).

La prévision des fautes est conduite en réalisant une évaluation du comportement du système en prenant en compte les occurrences et les activations de fautes. L'évaluation peut être **qualitative** (pour identifier, classifier et ranger les modes de défaillances) ou **quantitative** (pour évaluer en terme de probabilités l'étendue pour laquelle des attributs sont satisfaits).

1.2.2 Tolérance aux fautes dans le domaine avionique

Les systèmes avioniques tels que décrits dans la section 1.1 sont dotés de mécanismes de tolérance aux fautes pour réduire au minimum les risques de défaillances qui pourraient causer de nombreux dégâts pouvant aller jusqu'à des pertes humaines. Nous détaillerons ici deux mécanismes importants (mais d'autres existent) de tolérance aux fautes utilisés dans des avions : la redondance et la diversification fonctionnelle (les deux mécanismes utilisent le vote majoritaire pour prendre des décisions).

1.2.2.1 Utilisation de la redondance

La redondance est souvent utilisée comme mécanisme principal de la tolérance aux fautes pour améliorer la fiabilité en utilisant des ressources alternatives. La redondance peut être logicielle, matérielle ou temporelle. Nous allons détailler trois types de redondance : statique, dynamique et hybride.

La redondance statique utilise le masquage de fautes en réalisant un vote majoritaire entre les différentes répliques. Ainsi, si la majorité des répliques fonctionne correctement, une faute dans une minorité des répliques n'affectera pas le fonctionnement du système global. Cela suppose que les défaillances sont statistiquement indépendantes.

La redondance dynamique utilise la reconfiguration du système (action de modifier la structure d'un système qui a défailli, de telle sorte que les composants non-défaillants permettent de délivrer un service acceptable, bien que dégradé [L$^+$96]). C'est la majorité des répliques qui décident de désactiver ou non une réplique défaillante. Mais avant, le système doit déterminer si la défaillance est permanente ou temporaire. Ensuite, il établit quelles fonctions sont nécessaires pour poursuivre l'exécution du programme, il détermine si une évaluation des dommages est nécessaire et si une isolation de fautes ou une reconfiguration sont nécessaires.

La redondance hybride est la combinaison de la redondance statique et la redondance dynamique. C'est-à-dire que le masquage de fautes est appliqué en plus d'une

reconfiguration du système.

1.2.2.2 Diversification fonctionnelle

La diversification fonctionnelle est une technique qui utilise des variantes (les variantes correspondent à des répliques) et un décideur. C'est le décideur qui va commander l'ordre de désactiver ou non une variante. Cette technique est destinée à tolérer des fautes de conception logicielles. Il existe trois approches pour la diversification fonctionnelle : les blocs de recouvrement, la programmation en N-versions et la programmation N-autotestable [L$^+$96].

Avec **les blocs de recouvrement** [Ran75], le décideur teste les variantes de façon séquentielle. La variante primaire est exécutée en premier lieu. Si une défaillance est constatée, une autre variante (secondaire) est exécutée à la place de la primaire. Il est alors nécessaire de restaurer l'état de la première exécution pour l'exécution de la variante secondaire. Une défaillance peut être provoquée par une faute logicielle telle que : division par zéro, excès de limite de temps, *buffer overflow*, etc.

Avec **la programmation en N-versions** [Avi85], les variantes (ici appelées versions) et le décideur effectuent un vote majoritaire sur les résultats de toutes les versions. Par exemple, ce type de diversification fonctionnelle a été implémenté par Boeing pour le 777 en utilisant trois versions conçues par trois constructeurs différents (Intel, Motorola et AMD) ou en 4 versions pour les A330/A340 de Airbus. Pour Airbus, cette technique est utilisée depuis les premiers A300 (années 1970) avec des exécutions parallèles de deux variantes qui s'arrêtent quand une comparaison des résultats indique une différence [LABK90].

La programmation N-autotestable [LABK90] utilise des variantes en parallèle avec des mécanismes de détection d'erreurs. Un composant autotestable est un composant qui contient un mécanisme capable de contrôler si les propriétés entre ses entrées et ses sorties sont vérifiées. Si cela n'est pas le cas, un signal d'erreur est alors délivré au composant pour qu'il soit capable de se suspendre de lui-même et pour qu'une autre variante puisse continuer l'exécution.

Après avoir présenté les différents mécanismes de tolérance aux fautes, nous donnons dans la suite de ce chapitre, un aperçu les différentes normes utilisées dans les logiciels et matériels critiques.

1.2.3 Développement de logiciels critiques sûrs

Le développement logiciel des applications avioniques est réalisé en respectant un ensemble de normes et recommandations qui permettent de développer des systèmes critiques (logiciels et matériels) et de fournir des moyens de certifications. Le processus d'évaluation de la sécurité d'un système avionique a pour but de vérifier la conformité de la conception avec les exigences de navigabilité spécifiées par les autorités. Ces autorités sont l'EASA (*European Aviation Safety Agency*) pour l'Europe et la FAA (*Federal Aviation Administration*) pour les Etats-Unis. Les organismes suivants sont responsables de la rédaction des recommandations et normes : SAE (*Society of Automotive Engineers*),

RTCA (*Radio Technical Commission for Aeronautics*) pour les Etats-Unis et EUROCAE (*European Organisation for Civil Aviation Equipment*) pour l'Europe. La SAE[8] fournit les documents de type ARP (*Aerospace Recommended Practice*) ; la RTCA Inc.[9] fournit les documents de type DO et l'EUROCAE[10] fournit les documents de type ED. Nous pouvons également citer l'ARINC[11] (*Aeronautical Radio INCorporated* qui est une entreprise créée en 1929 par plusieurs compagnies aériennes et constructeurs aéronautiques américains et rachetée en août 2013 par Rockwell Collins. Elle définit de nombreuses recommandations pour huit types d'industries : l'aviation, les aéroports, la défense, le gouvernement, les services de santé, les réseaux, la sécurité ainsi que les transports. L'ARINC a notamment conçu dès 1978, le système ACARS (*Aircraft Communications Addressing and Reporting System*) qui permet d'effectuer encore aujourd'hui des communications entre les stations au sol et les avions en utilisant la radio (communication VHF[12] ou HF[13]) ou le satellite. Les organismes européens et américains travaillent en collaboration pour réaliser des documents équivalents, c'est pour cette raison que les noms des recommandations contiennent deux références différentes (DO et ED). Ces recommandations ARP, DO et ED sont devenues des normes avioniques au fil des années.

Quatre principales normes sont utilisées pour le développement logiciel critique. L'ARP 4754A/ED-79 (*Guidelines for Development of Civil Aircraft and Systems*) décrit un processus de développement des systèmes avioniques et inclut des études pour permettre leur certification. L'ARP 4761/ED-135 (*Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*), quant à elle, décrit un ensemble de méthodes d'analyse et d'évaluation de la sécurité-innocuité des systèmes avioniques à des fins de certification. Ces deux normes sont en général utilisée conjointement. Le DO-178B/ED-12C (*Software Considerations in Airborne Systems and Equipment Certification*) définit les contraintes de développement liées à l'obtention de la certification de logiciel avionique et le DO-254/ED-80 (*Design Assurance Guidance for Airborne Electronic Hardware*) est son pendant pour le développement d'équipement matériel électronique.

La figure 1.5 présente les liens entre ces différentes normes et recommandations.

1.2.3.1 Guide de développement pour l'aviation civile – ARP 4754A

La recommandation ARP 4754A (datant de décembre 2010) définit les processus de développement des avions civils et des systèmes critiques. Elle intègre notamment : la détermination et validation des exigences, le processus de certification, la gestion de la configuration, et la vérification de l'implémentation. Elle est utilisée conjointement avec l'ARP 4761 *"Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment"* qui définit les méthodes d'évaluation de la

[8] `http://www.sae.org`
[9] `http://www.rtca.org`
[10] `http://www.eurocae.org`
[11] `http://www.arinc.com`
[12] *Very High Frequency.*
[13] *High Frequency.*

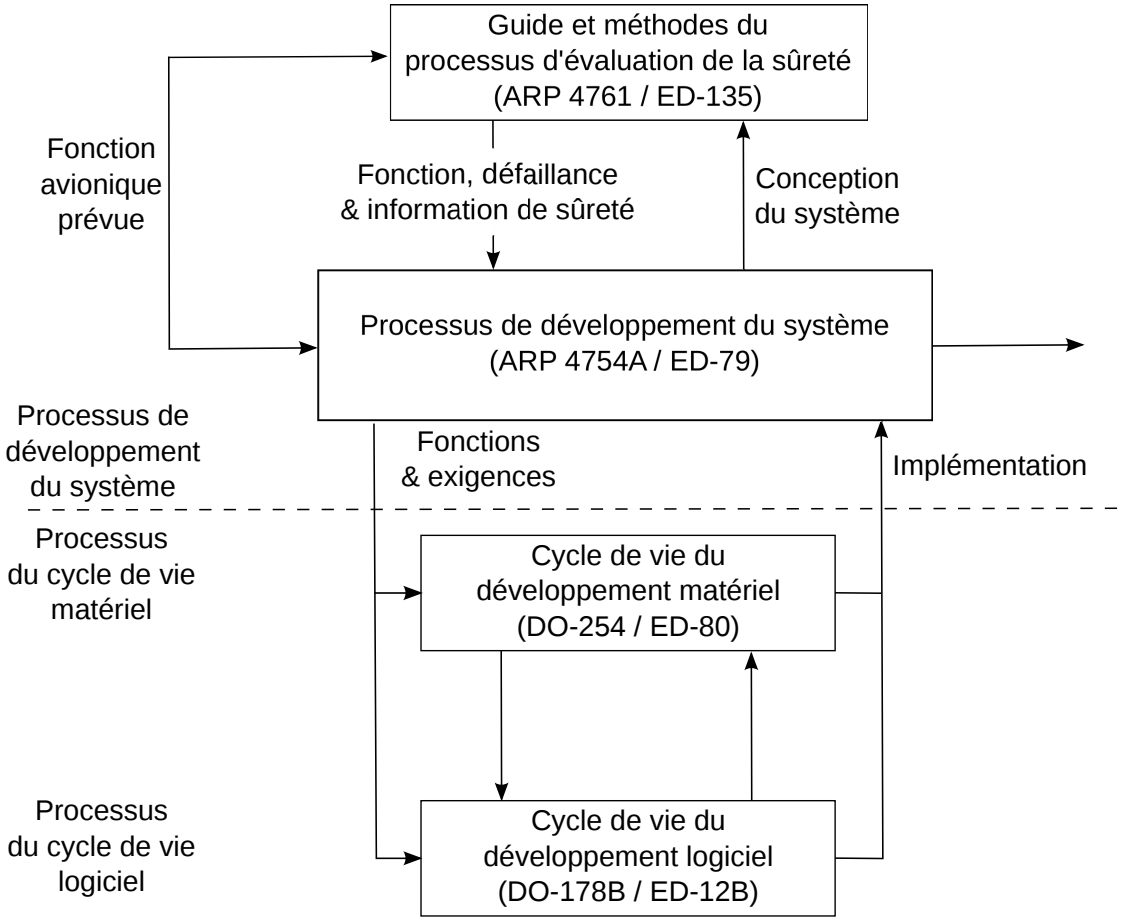

FIG. 1.5 – Guide de conception et techniques de certification (ARP 4754A, [arp10])

sécurité-innocuité des systèmes critiques et permet d'assigner les niveaux d'assurance de conception. La recommandation ARP 4754A permet de valider que les exigences sont correctes et complètes avec des méthodes de haut niveau sans faire de différences entre le logiciel et le matériel (qui est faite par ailleurs dans les recommandations DO-178C pour le logiciel et DO-254 pour le matériel).

1.2.3.2 Méthodes d'évaluation de la sûreté pour l'aviation civile – ARP 4761

Cette recommandation ARP 4761 définit plusieurs méthodes et techniques d'analyse de sûreté de fonctionnement sur le système, dont : *Functional Hazard Analysis* (FHA), *Preliminary System Safety Analysis* (PSSA), *System Safety Analysis* (SSA), *Fault Tree Analysis* (FTA), *Failure Modes and Effects Analysis* (FMEA). Les systèmes avioniques peuvent avoir des niveaux de criticité différents suivant leurs applications et donc leurs

développements imposent plus ou moins de vérifications/recommandations.

L'analyse de risques fonctionnels (FHA) permet d'identifier les défaillances potentielles du système et les conséquences de ces défaillances en attribuant un niveau de DAL (*Design Assurance Level*) auquel est associé une probabilité par heure qu'une défaillance ne survienne. Les cinq niveaux de DAL sont les suivants :

- Niveau A : Un dysfonctionnement du système ou sous-système étudié peut provoquer un problème catastrophique, la sécurité du vol ou de l'atterrissage est compromis et le *crash* de l'avion possible.
- Niveau B : Un dysfonctionnement du système ou sous-système étudié peut provoquer un problème majeur entraînant des dégâts sérieux voire la mort de quelques occupants.
- Niveau C : Un dysfonctionnement du système ou sous-système étudié peut provoquer un problème sérieux entraînant par exemple une réduction significative des marges de sécurité ou des capacités fonctionnelles de l'avion.
- Niveau D : Un dysfonctionnement du système ou sous-système étudié peut provoquer un problème ayant des conséquences mineures sur la sécurité du vol.
- Niveau E : Un dysfonctionnement du système ou sous-système étudié peut provoquer un problème sans effet sur la sécurité du vol.

Pour le niveau DAL A (niveau catastrophique), il est considéré comme atteint si la probabilité d'activation est inférieur à 1×10^{-9} par heure, alors que pour le niveau DAL D (niveau mineur), le seuil maximum est de 1×10^{-3} par heure, et pour le niveau E, aucun seuil n'est imposé.

L'analyse PSSA quant à elle, examine les conditions de défaillance de la FHA et décrit comment la conception du système devra respecter les engagements définis. Plusieurs techniques peuvent être utilisées pour le PSSA comme des arbres des fautes (FTA), des chaînes de Markov, etc.

1.2.3.3 Conception et certification de logiciels – RTCA DO-178C

La norme DO-178C qui est disponible depuis janvier 2012 est une mise à jour du DO-178B (qui datait de 1992). Elle décrit les objectifs pour les processus de cycle de vie des logiciels, les activités et les études de conception pour accomplir les objectifs et prouver que les objectifs ont été satisfaits. C'est en fait un modèle pour développer des logiciels de haute fiabilité. Il est donc important d'utiliser ces méthodes pour obtenir du code de confiance. Le nombre d'objectifs à atteindre dépend du DAL (voir 1.2.3.2) défini pour le logiciel analysé. Parmi les modifications apportées dans la version C, il y a notamment la volonté d'implémenter des approches qui peuvent changer avec la technologie, ce qui permet d'être compatible avec la modularité de l'architecture IMA. De même, quatre suppléments à ce standard ont été ajoutés : le DO-330 intitulé "*Software Tool Qualification Considerations*" pour la qualification d'outils avioniques, le DO-331 intitulé "*Model-Based Development and Verification Supplement to DO-178C and DO-278*" pour utiliser le développement basé sur les modèles [SPHP02], le DO-332 intitulé "*Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-*

278A" pour les conseils sur les technologies orientées objets, le DO-333 intitulé "*Formal Methods Supplement to DO-178C and DO-278A*" pour utiliser les méthodes formelles si nécessaire.

1.2.3.4 Conception et développement matériel – RTCA DO-254

Ce standard, qui date d'avril 2000, permet de concevoir et développer des équipements matériels complexes fiables pour des systèmes avioniques. Il est composé de plusieurs sections : cycle de vie de la conception matérielle, processus de planification, validation et vérification, gestion de la configuration, certification, etc. Les objectifs de ce standard sont identiques à ceux vus précédemment dans DO-178C, la seule différence étant qu'ici, il cible le matériel et non le logiciel.

1.2.4 Sécurité-immunité dans les standards avioniques

Les standards précédents décrivent les développements de systèmes critiques sûrs, pour empêcher que des fautes physiques et non des fautes d'interaction (voir sous-section 1.2.1.2) causent de graves conséquences. Ces standards sont donc plutôt orientés sécurité-innocuité. Cependant, il est intéressant de se pencher également sur la sécurité-immunité pour contrer d'éventuelles attaques informatiques car comme nous l'avons montré en début de ce chapitre, le nombre d'attaques informatiques augmente. Les pirates informatiques ont notamment aujourd'hui une plus grande surface d'attaques avec les systèmes avioniques plus récents car ils offrent plus de connectivité (plus d'interactions entre les systèmes et ressources) et ils utilisent moins d'équipements spécifiques et davantage de COTS (donc potentiellement moins protégés). Dans cette sous-section, nous nous intéressons à la sécurité-immunité et aux standards qui abordent ce sujet.

Des recommandations ont été rédigées par la FAA (§129.25 *Airplane Security*, §128.28 *Flight Deck Security* et §25.795 *Security Considerations*) mais elles n'adressent spécifiquement ni la sécurité des réseaux ni la sécurité des données. Nous détaillons ici trois standards, dont deux concernent plus particulièrement le domaine avionique (ARINC 811 et RTCA DO-326), qui abordent la sécurité au sens de sécurité-immunité (pour faire face aux malveillances).

On peut noter qu'il existe également des travaux qui s'intéressent aux interactions entre sécurité-innocuité et sécurité-immunité en proposant par exemple des approches développées harmonisées. On peut citer par exemple [DKCG99, B$^+$12a, B$^+$12b].

1.2.4.1 Common Criteria for Information Technology Security Evaluation – ISO/IEC 15408

Les critères communs correspondent à une norme internationale ISO/IEC 15408 qui permet de certifier des systèmes de sécurité. Cette norme n'est pas spécifique aux systèmes avioniques mais elle peut être appliquée à certains systèmes avioniques si besoin. Elle permet notamment d'assigner un niveau de certification *Evaluation Assurance Level* (EAL) entre 1 et 7 (du moins exigeant au plus exigeant en terme de conception, tests et

vérifications). Les produits certifiés ne garantissent pas l'absence de vulnérabilités mais seulement que des objectifs de tests et de vérifications ont été atteints. Par exemple, les produits tels que Microsoft Windows Server 2003 et Windows XP ont été certifiés en 2005 et sont évalués EAL4+, alors que nombreuses vulnérabilités ont été détectées depuis cette date : 629 vulnérabilités pour Windows XP et 532 vulnérabilités pour Windows Server 2003 [You].

1.2.4.2 Sécurité des systèmes d'information des systèmes commerciaux – ARINC 811

Le document ARINC 811 a été publié en 2005 par l'ARINC pour faciliter la compréhension de la sécurité de l'information et pour développer des concepts de sécurité opérationnelle des avions [ari05]. Même s'il a pour but de protéger les informations sensibles, ce document n'est pas directement utile pour se protéger contre une attaque informatique, ou pour éviter l'exploitation de vulnérabilités présentes dans des applications (critiques ou non). Il définit le cycle de vie de la configuration de l'avion, les modes des systèmes (opérationnel normal, opérationnel non-normal et maintenance), les rôles de sécurité (*Ground Security Administrator*, *Ground User*, *Onboard Cockpit User*, *Maintenance User*, etc.) qui permettent de définir des groupes de personnes auprès des compagnies aériennes, les objectifs de sécurité de l'information, etc. La figure 1.6 présente les différents domaines de réseaux dans un avion.

- 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). 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 défaillant.
- 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).

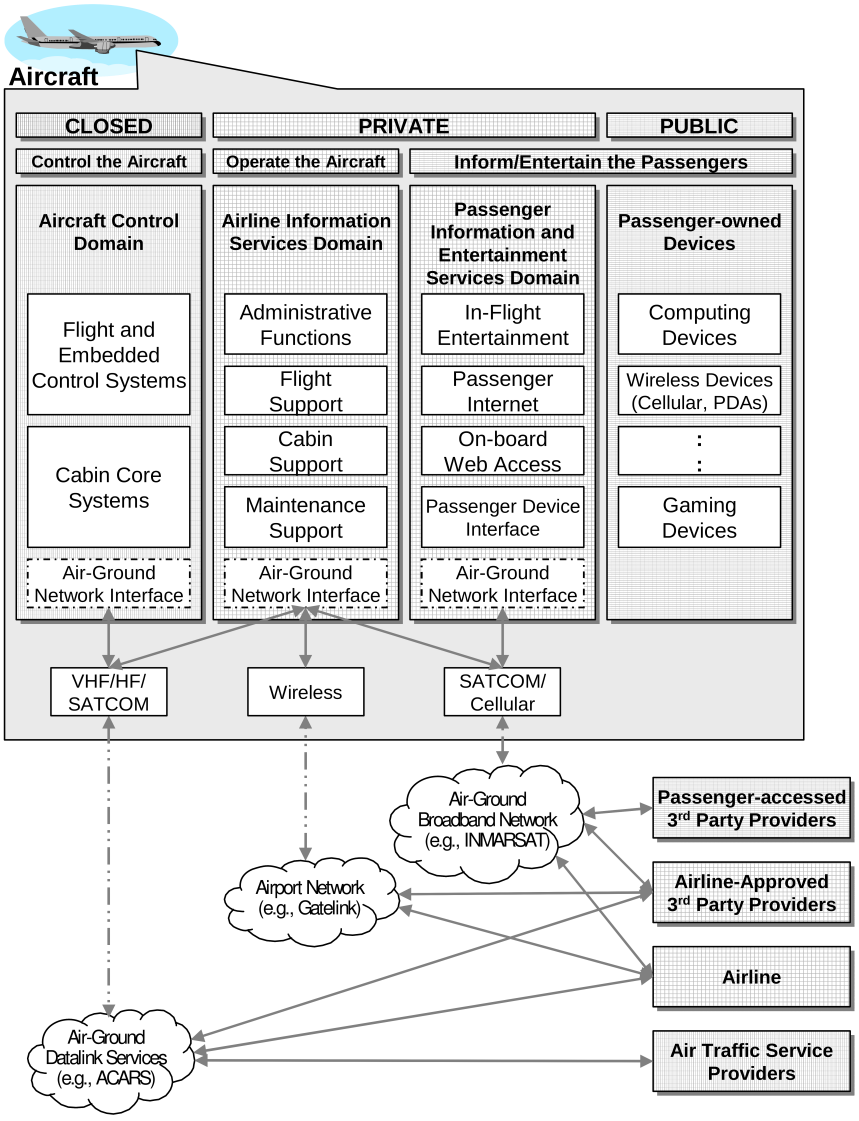

FIG. 1.6 – Les domaines de réseaux dans un avion (ARINC 811).

1.2.4.3 Processus de sécurité de la navigabilité – RTCA DO-326

La recommandation RTCA DO-326 intitulée "*Airworthiness Security Process Specification*" publiée en décembre 2010 [do310] permet d'améliorer la sécurité (au sens de sécurité-immunité) à bord. Ce standard a pour objectif de sécuriser les quatre éléments suivants :

- les connexions externes aux fournisseurs de services non-gouvernementaux (services de l'équipage de vol, services de maintenance) ;
- les nouveaux services de divertissement (isolation entre les systèmes de contrôle de l'avion, partage de ressources) ;
- les interfaces pour les média portables (incluant les ordinateurs portables) ;
- l'insertion des équipements externes.

Deux autres documents sont en cours de préparation pour compléter cette recommandation DO-326 : DO-YY3/ED-203 (*Security Methods and Considerations*) et DO-YY4/ED-204 (*Security Guidance for continuing airworthiness*). Le document ED-203 permet d'assurer par des méthodes et des recommandations, que les évaluations de risques de sécurité sont rigoureux, équilibrés et justes. Il permettra également de s'assurer par des pratiques d'assurance de développement que les contre-mesures sur l'avion sont implémentées correctement. Le document ED-204 permet de s'assurer que les contre-mesures sont fonctionnelles et qu'elles sont maintenues correctement. Une mise à jour de cette recommandation sera publiée courant 2014 (sous le nom de DO-326A) pour améliorer celle existante. La DO-326 était à l'origine conçue pour être un supplément de l'ARP 4754 orienté sécurité, mais la publication de ARP 4754A en 2010 impliquait trop de changements pour être seulement un supplément, donc un nouveau document est nécessaire. Ce standard DO-326 est actuellement en cours de révision pour être plus en phase avec les évolutions actuelles.

Malgré une volonté évidente de sécuriser les infrastructures avioniques, il est possible que des vulnérabilités subsistent et puissent être exploitées si des mécanismes de sécurité supplémentaires ne sont pas utilisés. Nous évoquons ce point dans la section suivante.

1.3 Problématique de la sécurité-immunité dans les systèmes avioniques

Comme nous l'avons vu précédemment (voir sous-section 1.1.2), les systèmes IMA permettent d'offrir une connectivité accrue, un partage des ressources entre les applications (potentiellement de criticité différente) d'un même calculateur, et favorisent l'utilisation de COTS. Ces évolutions sont des atouts (réduction des coûts de développement, des coûts de maintenance, du poids) mais du point de vue de la sécurité, la surface d'attaque pour un programme malveillant en est décuplée. En effet, si par exemple, le partage de ressources n'est pas suffisamment protégé, une partition malveillante peut lire des données d'une autre partition voire même les modifier et ainsi potentiellement provoquer de graves conséquences. Nous présentons dans les sous-sections suivantes les problèmes de sécurité liées à chacune de ces évolutions.

1.3.1 Connectivité accrue entre les applications

La connectivité à bord d'un avion peut prendre diverses formes. Elle concerne à la fois les communications entre différentes applications distantes, à travers un support de communication, qu'il soit filaire (ARINC 429, AFDX) ou non (VHF, satellite et Wifi). Elle concerne également les communications entre applications d'un même calculateur (partitions sur un même module). Par exemple, l'utilisation d'iPADs a été autorisée par la FAA depuis 2011 [DO11]. Ces iPADs peuvent être utilisés comme des EFB (*Electronic Flight Bag*) portables. A terme, la documentation avion papier anciennement nécessaire (représentant une masse de 15 kg environ) pourra donc être supprimée.

Un autre exemple est l'utilisation d'appareils électroniques en fonctionnement pendant le décollage et l'atterrissage des avions, qui est possible depuis 2014 [lem13]. Cette évolution peut comporter des risques si la sécurité n'est pas maîtrisée. La figure 1.7 représente les domaines de réseaux présents dans un avion décrits dans le standard ARINC 811.

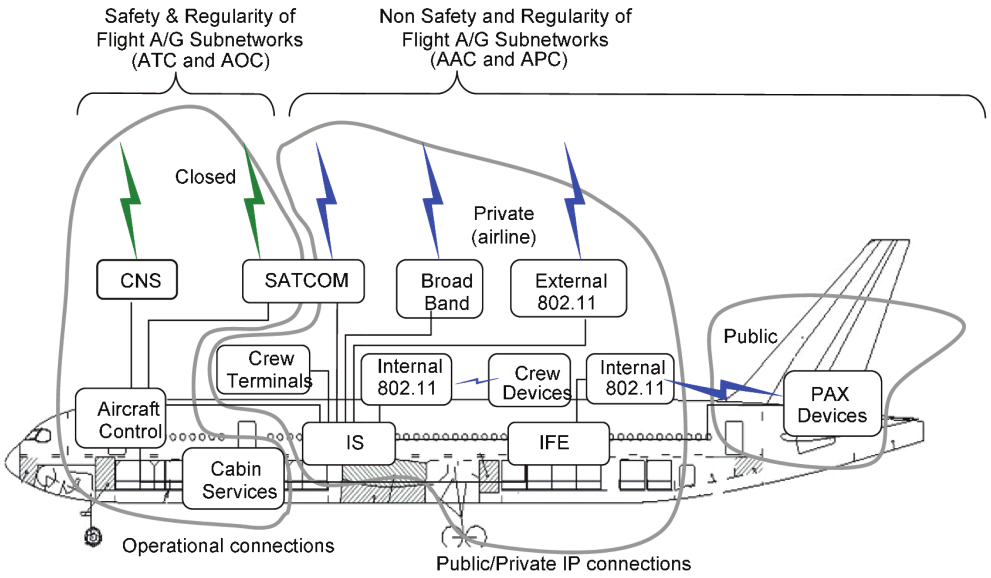

FIG. 1.7 – Les domaines de réseaux dans un avion (extrait de [ari05]).

Cette norme propose de décomposer les communications et équipements en trois catégories : publique, privée et fermée. La catégorie publique correspond aux appareils des passagers, comme les téléphones, ordinateurs portables, tablettes, etc. La catégorie privée correspond aux équipements de la compagnie aérienne comme les calculateurs fournissant l'accès à Internet, au Wifi, les systèmes de divertissement (IFE[14]). La caté-

[14] *In-Flight Entertainment.*

gorie fermée correspond au domaine où les informations sensibles circulent et ne doivent pas être accessibles par un tiers. Cela concerne notamment la communication par satellite (SATCOM) et par radio (VHF). Des règles précises de filtrage des communications entre chaque catégorie doivent être définies. Les équipements des passagers étant totalement imprévisibles, il faut être capable de limiter leurs accès pour éviter les risques de perturbations.

A l'heure actuelle, nous savons déjà qu'il est possible de perturber avec des ondes électromagnétiques de l'avion [Bro13]. Une étude de la IATA (*International Air Transport Association*) réalisée entre 2003 et 2009 sur 125 opérateurs a révélé que 75 cas de perturbations ont été détectés dont : des problèmes de communication, problèmes dans les contrôles de vol, des affichages défaillants, etc.

La maîtrise des communications entre différentes applications sur un système avionique est donc fondamentale du point de vue de la sécurité.

1.3.2 Partage de ressource entre applications de différents niveaux de criticité

Un autre avantage de l'architecture IMA est le partage facilité de ressources entre applications avec différents niveaux de criticité. Ce principe réduit les coûts de matériels mais la proximité physique des informations entre les applications entraîne un risque de compromission élevé. Pour cette raison, le partitionnement est utilisé. Il permet de restreindre les accès mémoire puisque chaque partition utilise une zone de la mémoire définie statiquement pendant la configuration. La figure 1.8 illustre un exemple simple de partage d'informations entre applications.

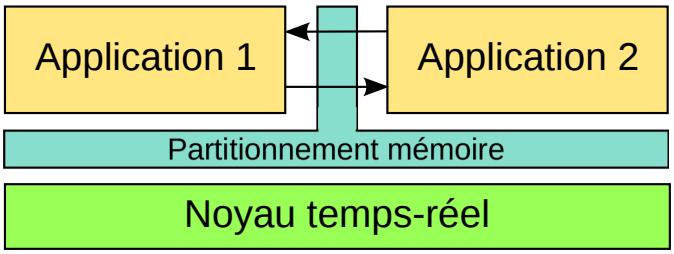

FIG. 1.8 – Partage simple entre 2 applications d'un module.

Dans la figure 1.8, nous avons deux applications sur le même module qui communiquent entre elles, le partitionnement permettant de contrôler les accès aux informations. Ce partage sur un système temps-réel doit être opéré de façon à ce que les données soient accédées de façon périodique ou apériodique mais elles ne doivent jamais être accessibles à deux applications en même temps pour éviter des accès concurrents à la même ressource.

Il est donc nécessaire de s'assurer que ce mécanisme de partitionnement soit efficace de manière à empêcher une application disposant d'un niveau de criticité faible de perturber le comportement d'une application de niveau de criticité élevé.

1.3.3 Utilisation de composants COTS

Aujourd'hui, l'utilisation de composants COTS (*Commercial-Off-The-Shelf*) représente plus de 95% de tous les composants de l'avion [Ska13] et ces composants n'ont pas forcément les mêmes exigences de conception et de développement que les composants avioniques spécifiques. Les fournisseurs de logiciels et matériels COTS sont susceptibles à tout moment de faire des modifications dans leurs logiciels et composants sans pour autant qu'elles soient validées à nouveau pour être embarquées dans l'avion. L'adoption d'une mise à jour d'un COTS pour utilisation dans un avion nécessite donc systématiquement un processus de validation complet.

Les composants COTS sont cependant acceptés à bord car d'une part, ils sont conçus par des fournisseurs choisis et approuvés par les constructeurs d'avions, et d'autre part, parce que ces composants ont été évalués et acceptés vis-à-vis d'un cahier des charges précis. Ces démarches réduisent considérablement a priori le risque que ces composants soient intentionnellement corrompus. Cependant, les composants de faible criticité sont évalués à un niveau inférieur à celui des composants de forte criticité ; il est donc toujours possible que des vulnérabilités subsistent.

1.3.4 Conclusion

Les évolutions de l'architecture des systèmes avioniques que nous venons de présenter (connectivité, partage des ressources et utilisation des COTS), augmentent la surface d'attaques possible et par conséquent, le risque potentiel de malveillance. Améliorer la sécurité des systèmes embarqués dans les avions est donc un problème important pour l'industrie aéronautique et il est fondamental aujourd'hui d'adapter les mécanismes de protection à l'évolution des architectures de leurs systèmes d'information.

Des cyber-attaques contre ces systèmes ont déjà été réalisées. En octobre 2011, des virus ont été détectés dans une flotte de drones américains Predator et Reaper [Mun11]. En avril 2013, des compagnies aériennes et constructeurs d'avion se sont intéressés de près à Hugo Teso, après des révélations sur la faible sécurité des communications dans les protocoles avioniques ADS-B[15] à ACARS[16]. Lors d'une démonstration sur un logiciel d'entraînement non avionable, il a voulu montrer qu'il était possible d'obtenir un accès au *Flight Management System* (FMS) [Tes13]. Même si sa démonstration n'était pas concluante, le fait qu'il l'ait présenté publiquement a été pris très au sérieux par l'ensemble de la communauté aéronautique, autorités et constructeurs.

Nous allons maintenant décrire dans la prochaine section les différentes approches existantes pour protéger les systèmes avioniques efficacement face à des malveillances.

1.4 Différentes approches pour améliorer la sécurité-immunité

Pour assurer la sécurité-immunité des systèmes avioniques, il est nécessaire d'appliquer ou d'adapter des méthodes et techniques qui ont prouvé leur efficacité pour la

[15] Automatic Dependent Surveillance-Broadcast.
[16] Aircraft Communication Addressing and Reporting System.

protection de systèmes informatiques “classiques” :

- les outils et mécanismes de sécurité (pare-feux, contrôle d'accès, système de détection d'intrusion, etc.) ;
- les méthodes formelles pour la spécification, le développement, la vérification et la certification ;
- l'analyse de vulnérabilités avec les contre-mesures associées, appliquées au plus tôt dans le cycle de développement.

Nous allons présenter maintenant ces trois approches dans les prochaines sous-sections.

1.4.1 Intégration de mécanismes de sécurité adaptés

Assurer la sécurité-immunité des systèmes peut en partie être réalisé en utilisant des outils et mécanismes de sécurité classiques. Plusieurs types d'outils existent : les *firewalls*, le contrôle d'accès, les systèmes de détection d'intrusion (*Intrusion Detection System* ou IDS), l'authentification, le chiffrement, etc. Ces outils ont montré leur efficacité pour la protection face aux attaquants sur les systèmes classiques. En revanche, ils doivent être adaptés pour satisfaire les contraintes des systèmes avioniques. Les deux contraintes principales sont le respect des délais d'exécution et le partitionnement spatial et temporel des partitions.

A titre d'exemple, dans un système classique, un IDS peut être déployé sur une machine de manière à observer à la fois les communications concernant les applications de la machine et les communications entre deux autres machines. Un IDS peut ainsi se baser sur ces observations pour remonter des alertes correspondant à l'identification d'une tentative d'attaque. Il doit donc pouvoir avoir accès à l'ensemble des communications. Par contre, sur un système avionique, une application peut observer uniquement les communications la concernant. Ce dispositif serait donc dans l'incapacité de détecter des tentatives d'intrusion concernant les autres applications. Donc, les IDS employés dans les réseaux d'entreprises ne peuvent pas être déployés en l'état sur les systèmes embarqués.

Il est relativement simple de mettre en place les mécanismes de chiffrement dans les systèmes avioniques. Par contre, un inconvénient majeur pour ces mécanismes concerne le temps de chiffrement qui peut être relativement long (surtout pour des calculs concernant certains chiffres assymétriques tels que le RSA). De la même manière, pour les mécanismes de contrôles d'accès et d'authentification, des vérifications doivent être réalisées à chaque accès aux ressources, ce qui peut s'avérer également particulièrement coûteux.

1.4.2 Utilisation de méthodes formelles

Les méthodes formelles sont utilisées par les avionneurs pour vérifier le fonctionnement des applications et des systèmes [S$^+$09b]. Par exemple, Airbus utilise ces méthodes depuis 2001 pour l'A380. La norme RTCA DO 178C propose notamment une annexe concernant l'utilisation de ces méthodes.

Les méthodes formelles fournissent une structure mathématique avec laquelle il est possible de s'assurer que le modèle d'un système satisfait bien un cahier des charges formel. Mais les méthodes formelles ont aussi des limitations [LA95], les spécifications formelles peuvent ne pas être efficaces car il n'y a pas de principe général sur la façon d'acquérir les exigences de l'utilisateur et comment les enregistrer en utilisant un langage formel de spécification. Le raffinement qui consiste à passer de l'abstraction d'une spécification à l'implémentation est une opération sensible qui peut engendrer de nouveaux problèmes si cette opération est mal réalisée. Les preuves formelles sont elles aussi sujettes à des problèmes suivant le raffinement qui est réalisé. De la même façon, la spécification peut ne pas être suffisamment détaillée pour le développeur ou alors les exigences de l'utilisateur peuvent être mal comprises par le concepteur/développeur. D'autres problèmes potentiels plus spécifiques aux systèmes sécurisés sont cités dans [JH08] : spécifications invalides ou partielles, définitions de propriétés trop vagues, implémentation trop "libre" pour le développeur qui amène à des interprétations personnelles.

Nous pouvons ajouter à cela que les méthodes formelles doivent se baser sur des hypothèses sur l'environnement d'exécution, soit parce que l'environnement d'exécution n'est pas encore disponible, soit parce que sa complexité nécessite de le modéliser de manière plus simple. La figure 1.9 illustre ce problème.

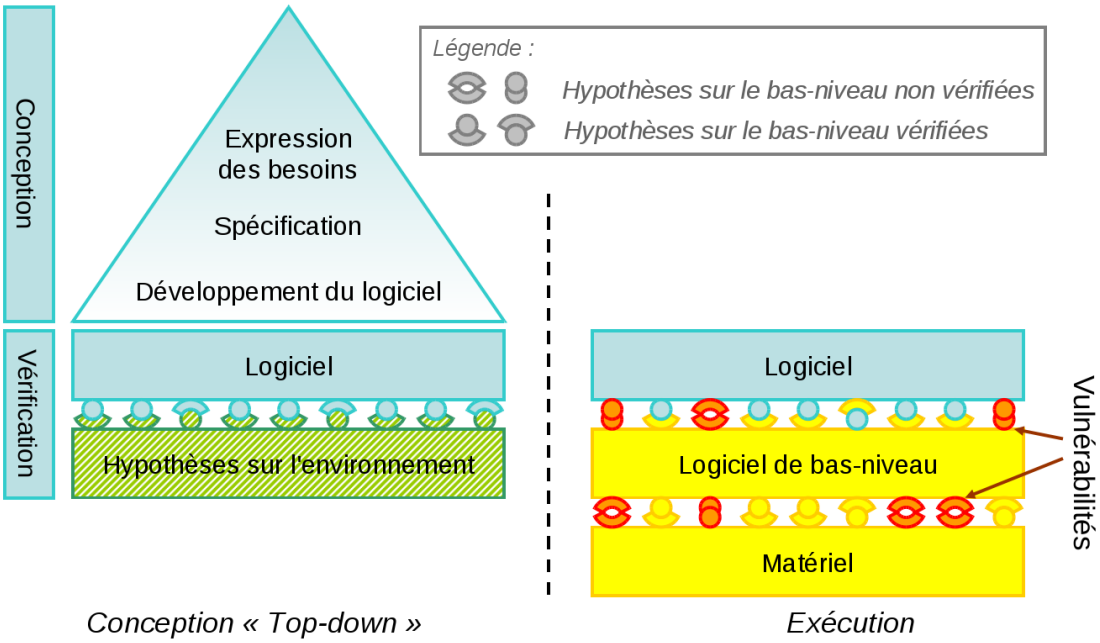

FIG. 1.9 – Visualisation des hypothèses de bas niveau.

La partie gauche de la figure représente les étapes nécessaires pour une conception ascendante et une vérification de l'application ("Conception *top-down*") et la partie droite représente l'exécution de ce même système. Pendant la vérification de la conception,

certaines hypothèses doivent être faites sur l'environnement dans lequel il sera exécuté. C'est à ce niveau que des problèmes peuvent subsister car ces hypothèses peuvent ne pas correspondre ou être légèrement différentes de la réalité dû à des changements non-prévus après la conception, des reconfigurations, etc. Ceci est en particulier vrai pour le matériel dont il est très difficile à priori de connaître parfaitement le fonctionnement. Si certaines hypothèses de bas niveau sont effectivement vérifiées, il est possible que d'autres ne le soient pas, ce qui peut conduire à la présence de vulnérabilités.

1.4.3 Analyse de vulnérabilités dans le processus de développement

La troisième approche consiste à analyser un composant (matériel, noyau temps-réel et applications) en vue de détecter la présence de vulnérabilités. Une vulnérabilité est une faute qui peut être exploitée pour corrompre une partie ou la totalité du système. Il est donc important de détecter ces vulnérabilités pour éviter des conséquences graves sur la sécurité du système.

Les analyses de vulnérabilités permettent de détecter les vulnérabilités mais elles sont généralement réalisées lorsque les systèmes sont en production pour étudier s'ils sont vulnérables à des attaques informatiques. L'inconvénient de cette démarche est que, lors de l'identification d'une vulnérabilité, la contre-mesure se limitera à une "rustine" locale appliquée au système. Cette contre-mesure ne pourra donc pas être intégrée directement dans l'architecture de ce système. Or, il est également possible et intéressant de réaliser une analyse de vulnérabilités au plus tôt dans le développement du noyau et des applications pour les détecter avant que le système soit en production. Ces analyses de vulnérabilités peuvent également être intéressantes pour identifier l'architecture logicielle la plus adaptée pour éviter l'introduction des vulnérabilités.

Nous avons décidé de développer cette approche pour la suite car c'est une approche innovante pour les systèmes critiques et elle peut donner de bons résultats. L'analyse de vulnérabilités réalisée tôt dans le processus de développement permettra de corriger ces vulnérabilités pour améliorer la protection globale du système embarqué. Dans la prochaine section, nous détaillons l'analyse de vulnérabilités dans les systèmes critiques embarqués.

1.5 Analyse de vulnérabilités dans les systèmes critiques embarqués

Nous nous intéressons dans cette section à l'analyse de vulnérabilités sur des systèmes classiques, pouvant être adaptée à des systèmes temps-réel. Pour les systèmes non temps-réel, les analyses de vulnérabilités sont souvent utilisées pour détecter si des maliciels sont présents, si des données peuvent être volées ou si le système peut être corrompu. Une fois détectée, il est nécessaire de trouver des moyens pour se protéger efficacement et ainsi éviter que cette vulnérabilité ne soit exploitée. Les analyses de vulnérabilités peuvent être réalisées de manière automatique ou de manière manuelle. Elles peuvent être également réalisées par des approches boîte noire ou par des approches boîte blanche.

Une analyse de vulnérabilités manuelle a l'avantage d'être plus approfondie mais cette analyse requiert en général des compétences très spécifiques que seul un expert humain est capable de mettre en pratique. De plus, le temps nécessaire pour faire cette analyse est long car il est très difficile de prendre en compte toutes les subtilités d'un système, étant donné sa complexité. Une analyse automatique permet en revanche une analyse plus rapide mais plus superficielle car les outils ne font pas de recherche approfondie contrairement à l'analyse manuelle. En effet, il est très difficile de développer des outils d'analyse automatique en leur permettant de traiter un large éventail de vulnérabilités. Cette difficulté est en partie due à la complexité des cibles des analyses. De nombreux travaux ont été réalisés pour évaluer la présence de vulnérabilités dans des systèmes, par différents moyens [IW08].

Les approches boîte blanche et les approches boîte noire se distinguent par les conditions dans lesquelles ces analyses sont réalisées. L'approche par boîte blanche consiste à avoir accès à toutes les informations du système allant du code source à la documentation et permet ainsi de comprendre exactement comment il fonctionne. L'approche par boîte noire consiste à ne pas avoir ces informations et donc à travailler sans aucune connaissance particulière sur le système.

Nous allons décrire ces deux types d'analyses dans les prochaines sous-sections 1.5.1 et 1.5.2 pour en comprendre le fonctionnement. L'approche que nous proposons sera ensuite décrite dans la sous-section 1.6.

1.5.1 Approche par boîte blanche

L'approche par boîte blanche consiste à avoir accès à des informations du système (code source, binaire et documentation éventuelle) pour réaliser l'analyse de vulnérabilités. Deux techniques différentes sont possibles pour réaliser cette analyse. L'une "statique" consiste à analyser le code source ou le fichier binaire mais sans exécuter le programme ciblé. L'autre technique est dite "dynamique", elle consiste à faire l'analyse pendant l'exécution du programme ciblé en observant les flux de données (entrées et sorties) soit en instrumentant le code (i.e. modifier le code avec des informations de debug) soit en le laissant tel quel. Cependant, si le programme a été modifié pour réaliser l'instrumentation, il se peut que son comportement soit différent du comportement dans l'environnement réel d'exécution. En général, des outils d'analyse de code sont utilisés pour détecter les vulnérabilités, ils analysent le code source pour détecter les erreurs courantes de débordement de buffers (*buffer overflows*), débordement d'entiers (*integer overflows*) ou encore de divisions par zéro.

Plusieurs outils d'analyse statique sont utilisés dans l'industrie, nous en présentons 4 :

- Caveat [B+02] est un outil de vérification formelle qu'utilise Airbus depuis 2002. Il permet d'analyser des programmes en C et propose deux fonctionnalités : les analyses de flux de contrôle et de données (automatique pour l'outil) et la preuve des propriétés spécifiées par l'utilisateur.

- Frama-C[17] permet d'analyser des programmes C et de s'assurer que le code source est conforme à une spécification formelle définie. Il est possible d'ajouter des plug-ins pour y ajouter des fonctionnalités. Ces plug-ins peuvent ensuite communiquer entre eux grâce à un langage de spécification commun appelé ACSL.
- Astrée[18], également utilisé par Airbus, permet de prouver l'absence de *Run-Time Error* (RTE) [C+07] en utilisant la théorie de l'interprétation abstraite [CC77].
- Ait[19] analyse un programme dans sa forme binaire pour calculer une borne supérieure du WCET des tâches du programme. Il permet de vérifier que les WCET calculés correspondent à ceux mesurés.

Ces outils sont utiles pour vérifier que le code (ou le fichier binaire) fait ce qu'il doit faire sans faute de développement mais des vulnérabilités peuvent encore subsister. Les analyses menées par ces outils peuvent donc être complétées par des analyses dites "approches par boîte noire".

1.5.2 Approche par boîte noire

L'approche par boîte noire consiste à réaliser l'analyse de vulnérabilités sans avoir une connaissance particulière du système. Elle repose généralement sur l'observation des interactions du composant analysé avec son environnement. Les techniques statiques et dynamiques pourraient être appliquées dans ces approches. Cependant, les techniques statiques ne sont pas pertinentes car elles imposent d'avoir accès soit au code source soit au binaire, avant l'exécution et elles ne permettent donc pas d'être dans un contexte d'approche par boîte noire. En revanche, les techniques dynamiques sont pertinentes car, pendant l'exécution, l'analyse de vulnérabilités peut être réalisée en analysant les flux d'entrées et de sorties (par exemple, vérification des limites d'un tableau en mémoire, etc.) sans avoir d'informations particulières sur le programme exécuté. Les différentes étapes nécessaires sont les suivantes :

- identifier les composants de l'environnement ;
- sélectionner des attaques susceptibles de fonctionner ;
- tester les attaques sélectionnées ;
- observer le comportement du système.

L'identification des composants de l'environnement permet de connaître l'ensemble des composants avec lesquels le composant à analyser peut interagir. Ces interactions sont réalisées par des entrées et des sorties. Souvent, les outils d'analyse dynamique adoptant une approche boîte noire se basent sur ces entrées et sorties pour tester les composants. La seconde étape correspond à un croisement entre les informations collectées dans la première étape avec une base de connaissance sur des classes d'attaques. En général, la connaissance de l'environnement est nécessaire pour tester des situations extrêmes. Ces classes d'attaques sont utilisées pour atteindre ces situations extrêmes. Pour

[17] `http://frama-c.cea.fr/`
[18] `http://www.astree.ens.fr/`
[19] `http://www.absint.com/ait/`

pouvoir être testées, ces classes d'attaques doivent être instanciées. Une classe d'attaque souvent employée est la génération de données aléatoires (*fuzzing*). Les attaques obtenues sont ensuite testées sur le composant, dans la troisième étape. Pour finir, le comportement du composant est analysé dans la dernière étape. Si, lors de l'exécution de l'attaque, une défaillance est identifiée (par exemple, le composant ne délivre plus de service), alors l'attaque est considérée comme réussie. Il faut aussi prendre en compte l'impact des mécanismes de tolérance aux fautes du composant sur le déroulement du processus de tests (par exemple un redémarrage du système conduira à l'arrêt du test). Les méthodes d'analyse dynamique ont deux défauts principaux [C+07] :

- elles peuvent prouver la présence d'erreurs mais pas leurs absences ;
- elles ne peuvent pas vérifier toutes les propriétés d'un programme (présence de code mort ou non terminaison).

1.6 Contribution de cette thèse : une méthodologie d'analyse de vulnérabilités pour des systèmes embarqués avioniques

Les sections précédentes décrivent les systèmes avioniques et les techniques de sûreté de fonctionnement utilisées dans ces systèmes pour éviter des conséquences graves sur les systèmes avioniques. Les normes et techniques permettant d'améliorer la sécurité-innocuité et la sécurité-immunité à bord de l'avion sont également présentées dans ces sections. Un focus particulier est réalisé sur l'analyse de vulnérabilités, qui constitue une de ces techniques. Habituellement, les analyses de vulnérabilités sont réalisées à la fin du développement et en phase d'exploitation et, en cas d'identification de vulnérabilités, ce sont des "rustines" qui sont appliquées localement, la plupart du temps sans essayer de repenser l'architecture globale de l'application. Effectivement, le temps restant pour produire le système ne permet pas de revenir sur les phases de conception et de modélisation du système. Réaliser les tests et découvrir des vulnérabilités pendant le développement à partir du moment où l'on dispose d'un premier prototype du système permet une toute autre approche. La découverte de certaines vulnérabilités, jugées critiques, peut amener à une modification de l'architecture globale de l'application, parce qu'il est encore temps de le faire sans que ce soit trop coûteux. On peut ainsi obtenir une architecture construite pour "intrinsèquement" éviter l'introduction des vulnérabilités identifiées, dès les premières phases de son développement.

Cette analyse de vulnérabilités et la proposition de contre-mesures adaptées au plus tôt dans le cycle de développement logiciel est au cœur de la contribution de cette thèse. Nous allons pour cela proposer une méthodologie d'analyse de vulnérabilités, basée sur trois grandes étapes :

- Dans un premier temps, il est fondamental de classifier l'ensemble des menaces qui peuvent peser sur les systèmes embarqués avioniques. C'est la contribution proposée par le chapitre 2. Dans ce chapitre, nous proposons de définir une classification la plus exhaustive possible des différentes attaques qui peuvent cibler les systèmes

embarqués avioniques. Bien sûr, certaines attaques sont similaires à des attaques que l'on peut rencontrer sur des systèmes informatiques usuels "du bureau", mais certaines sont spécifiques aux systèmes avioniques car elles peuvent tirer partie de certaines caractéristiques spécifiques de ces systèmes pour arriver à leur fin. Nous détaillons donc ces deux familles d'attaques dans notre classification.

- Une fois cette classification établie, une phase d'expérimentation sur le système cible doit avoir lieu. Ces expérimentations doivent faire en sorte de tester aussi exhaustivement que possible les attaques décrites dans la classification précédente. Bien évidemment, elles doivent être précédées d'une analyse précise du système cible et de son fonctionnement interne de façon à pouvoir parfaitement mener des attaques adaptées à ce système. Cela signifie que l'analyse doit tenir compte à la fois du logiciel mais aussi du matériel sur lequel le système s'exécute. C'est la raison pour laquelle il est fondamental que les expérimentations se déroulent sur une plateforme matérielle identique à la plateforme sur laquelle le système avionique doit réellement s'exécuter en opération. Le chapitre 3 présente un exemple de mise en œuvre de telles expérimentations sur un système avionique expérimental fourni par Airbus France. Ce chapitre commence donc par présenter en détail l'implémentation de ce système, ainsi que la plateforme sur laquelle il s'exécute. Ensuite, il présente l'ensemble des attaques qui ont été envisagées sur cette plateforme et décrit en détail celles qui ont été les plus pertinentes en terme de découvertes de vulnérabilités.
- Enfin, la dernière phase de notre méthodologie consiste à concevoir et implémenter des contre-mesures efficaces face aux vulnérabilités découvertes. C'est l'objet du chapitre 4. Les contre-mesures proposées dans ce chapitre sont de deux types. Nous y décrivons dans un premier temps les contre-mesures spécifiques aux vulnérabilités découvertes sur le système présenté dans le chapitre 3. Ensuite, nous essayons de généraliser ces contre-mesures, de façon à proposer des recommandations, aussi génériques que possible, à appliquer durant le développement de systèmes embarqués avioniques.

Chapitre 2

Caractérisation des attaques des systèmes embarqués

Introduction

Dans le chapitre précédent, nous avons justifié la nécessité d'améliorer et de réaliser systématiquement des analyses de vulnérabilités dans le processus de développement des systèmes embarqués avioniques de future génération. En effet, l'évolution des systèmes (ouverture, partage, utilisation de COTS) facilitent l'introduction de vulnérabilités et il est donc fondamental d'accompagner leur développement d'une analyse de vulnérabilités approfondie, au plus tôt, pour renforcer leur sécurité.

Il est nécessaire d'adopter une démarche rigoureuse pour mener cette analyse de vulnérabilités. Pour cela, une connaissance précise des attaques qui peuvent menacer ces systèmes est requise. C'est la raison pour laquelle ce chapitre propose une classification des différentes attaques qui peuvent cibler des systèmes embarqués critiques avioniques. Cette classification a fait l'objet d'une publication dans [DADN12].

Nous avons établi cette classification en faisant le raisonnement suivant. Un système embarqué critique a de multiples points communs avec un système informatique "classique" utilisé dans le cadre professionnel ou privé. Nous allons donc établir une première classification d'attaques inspirée des classifications connues des attaques sur les fonctionnalités de base des systèmes informatiques classiques, en les adaptant toutefois aux spécificités et aux contraintes du monde avionique. Nous appellerons cette catégorie *attaques ciblant les fonctionnalités de base*. Cependant, un système avionique embarqué possède des caractéristiques spécifiques très particulières pour des raisons de sûreté de fonctionnement (*safety*). Il inclut en particulier un ensemble de mécanismes de tolérance aux fautes. Or, ces mécanismes eux-mêmes peuvent être potentiellement la cible d'attaques et leur corruption peut s'avérer particulièrement intéressante pour un attaquant. Nous le montrerons dans la suite de ce chapitre. Nous avons donc défini une seconde catégorie d'attaques appelée *attaques ciblant les mécanismes de tolérance aux fautes*.

Ce chapitre est principalement consacré à la présentation de ces deux catégories d'attaques. A la fin de ce chapitre, nous présentons également le contexte et les hypothèses

d'attaques que nous avons considérés pour notre étude.

2.1 Attaques visant les fonctionnalités de base

Un système avionique embarqué, comme tout système informatique, utilise un processeur, de la mémoire, des périphériques, un système d'exploitation et des applications, et les mécanismes de gestion qui leurs sont associés. Nous avons ainsi classifié les attaques visant les fonctionnalités de base en huit catégories.

- le processeur ;
- la gestion de la mémoire ;
- les communications ;
- la gestion du temps ;
- la gestion des processus ;
- l'ordonnancement ;
- la cryptographie ;
- les fonctions ancillaires.

Les sous-sections suivantes présentent ces différentes catégories. Pour chaque catégorie, nous décrivons rapidement le fonctionnement du composant ciblé ou de la fonctionnalité ciblée et nous donnons le principe général des attaques que nous estimons possibles sur ce composant ou cette fonctionnalité. Nous nous inspirons des attaques connues dans le monde des PCs de bureau et de l'architecture x86, qui sont des architectures bien plus éprouvées par les attaquants que les architectures avioniques, du point de vue de la sécurité. Nous pouvons ainsi proposer un riche panorama d'attaques possibles, même si aujourd'hui, de telles attaques n'ont encore pas toutes vu le jour sur des architectures de type avionique. L'important, est d'être aussi exhaustif que possible afin de pouvoir anticiper au mieux l'occurrence de nouvelles attaques.

2.1.1 Attaques ciblant le processeur

Le processeur est le composant matériel principal d'un calculateur. Il exécute les différentes instructions. De nombreuses caractéristiques et améliorations sont incluses dans les processeurs modernes : multiples cœurs, multiples niveaux de mémoires caches, unité de gestion de la mémoire, unité de prédiction de branchement, interruptions, etc.

En particulier, les caches des processeurs sont utilisés pour réduire le temps moyen d'accès mémoire (car la mémoire SRAM (*Static Random Access Memory*) utilisée est beaucoup plus rapide que la DRAM (*Dynamic Random Access Memory*). En revanche, la SRAM ne peut contenir que très peu de données. Les systèmes actuels ont généralement trois niveaux de caches (L1, L2 et L3) placés de façon à optimiser les accès mémoire. Le premier niveau de cache (par exemple de 64 Ko) est au plus proche des cœurs du processeur. Il peut être partagé en deux caches, un pour les instructions et un autre pour les données (architecture Harvard) ou non (architecture von Neumann). Le second niveau de cache (par exemple de 128 Ko) est partagé entre les cœurs et est en général unifié (aucune séparation entre instructions et données). Le troisième niveau de cache se

situe après le cache L2. Il est donc plus lent mais contient plus de données (par exemple 8 Mo). Lorsque ces caches sont utilisés, les données de la mémoire y sont copiées pour être réutilisées plus tard.

Des mécanismes de cohérence des caches sont mis en œuvre pour que les caches contiennent les valeurs correctes aux mêmes moments. Des problèmes de sécurité liés à la gestion des caches dans les systèmes temps-réel ont déjà été mis en évidence [ZBR07] et des protections ont été proposées notamment en authentifiant et chiffrant les données [LAEJ11]. Des faiblesses concernant la confidentialité ont aussi été découvertes. Par exemple, des études montrent qu'il est possible de retrouver des clés secrètes cryptographiques en utilisant les caches [Aci07a, OST06, Aci06, Bon06, Aci07b, Per05, Ber05] ou le mécanisme de prédiction de branches BTB [ASK07]. La figure 2.1 illustre la hiérarchie des caches présents dans un système utilisant 4 cœurs.

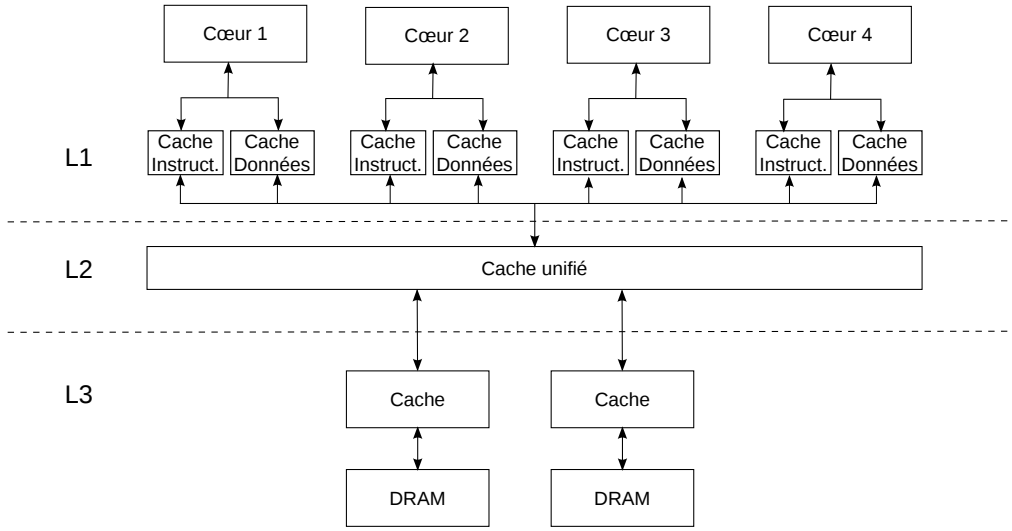

FIG. 2.1 – Hiérarchie des caches (architecture Harvard).

Des attaques peuvent aussi essayer de provoquer un arrêt inopiné du processeur, en le faisant exécuter des instructions non-définies ou non-documentées. Si un code malveillant est capable d'envoyer des instructions non-documentées au processeur, le comportement de ce dernier devient imprévisible. Cela peut provoquer, par exemple, le basculement du processeur dans un mode spécial (tel que un mode de maintenance) dont bénéficie automatiquement le code malveillant qui s'exécute. Il peut ainsi acquérir de nouveaux privilèges associés à ce mode. Une telle attaque peut provoquer un Déni-de-Service[20] (DoS). Par exemple, comme décrit dans [SPL95], les processeurs x86 peuvent avoir des comportements imprévisibles : les auteurs ont identifié 102 vulnérabilités potentielles affectant des versions différentes des processeurs 80x86. Il a d'ailleurs été montré que certains processeurs de la famille x86 disposaient d'instructions non documentés, c'est-

[20] *Denial-Of-Service.*

à-dire pour lesquels on ne connaît pas le comportement du processeur [Duf07]. Tenter d'exécuter ces instructions peut donc provoquer des comportements totalement imprévus et peut mener à la défaillance du système reposant sur ce processeur. Ainsi, des programmes tels que *Crashme*[21] générant un flot d'instructions aléatoires, ont déjà montré leur efficacité sur différents types d'architecture.

2.1.2 Attaques ciblant la gestion de la mémoire

Les attaques ciblant la mémoire correspondent à des attaques qui visent à modifier certaines régions de mémoire du calculateur. Le terme "mémoire" est à considérer au sens large : il inclut la mémoire centrale (RAM), mais aussi l'environnement d'exécution du processeur (tels que les registres par exemple) et les régions de mémoires incluses dans les contrôleurs d'entrées/sorties. Ces attaques peuvent être perpétrées depuis un programme s'exécutant sur le processeur en exploitant une vulnérabilité logicielle, mais aussi depuis un périphérique malveillant, abusant les mécanismes d'entrées/sorties tels que les accès directs à la mémoire (DMA ou *Direct Memory Access*).

2.1.2.1 Accès à la mémoire depuis le processeur

La façon la plus directe d'accéder à la mémoire consiste à utiliser le processeur. Normalement, ces accès sont limités à la mémoire attribuée au processus en cours d'exécution. Ces privilèges peuvent être étendus grâce à certaines fonctionnalités des systèmes d'exploitation. Par exemple, sous Linux, il est possible d'accéder à la mémoire du système d'exploitation au travers de l'utilisation des fichiers */dev/mem* ou */dev/kmem* ou au travers de certaines fonctionnalités matérielles. Ainsi Joanna Rutkowska, dans [RW09], a montré que le mode SMM (*System Management Mode*) des processeurs Intel peut être détourné pour obtenir un accès direct à toute la mémoire.

L'accès à la mémoire depuis le processeur peut être également réalisé en exploitant des vulnérabilités (telles que les débordements de tampons, les chaînes de formats, etc.), qui peuvent affecter tout type de composant logiciel (applications, système d'exploitation, gestionnaire de machines virtuelles). Des exemples de telles vulnérabilités ont déjà été recensés sur des systèmes tels que Windows [CC13], Mac [Art14] et Linux [Aed12], en particulier pour les plateformes x86. Ces vulnérabilités sont dûes à un manque de rigueur dans le codage des applications, en particulier l'absence de tests aux valeurs limites et l'absence de vérification lors d'allocation ou libération de mémoire. Les conséquences de l'exploitation de ces vulnérabilités peuvent être sérieuses puisqu'elles peuvent permettre à l'attaquant d'exécuter du code sur la machine cible, avec différents niveaux de privilège, en fonction du logiciel vulnérable exploité. En particulier, si une vulnérabilité de ce genre est présente non seulement dans les couches applicatives mais au sein du noyau du système lui-même (ce qui reste possible malgré le niveau de criticité du noyau), son exploitation permet alors à un attaquant d'exécuter du logiciel malveillant avec les privilèges du noyau, ce qui rend ces attaques particulièrement graves.

[21] `http://people.delphiforums.com/gjc/crashme.html`

2.1.2.2 Accès à la mémoire par les entrées/sorties

L'accès à la mémoire peut être réalisé par le mécanisme d'accès direct à la mémoire (DMA) piloté par des contrôleurs d'entrées/sorties. Ceux-ci permettent d'accéder à la mémoire centrale en lecture et en écriture mais aussi aux mémoires internes et registres d'autres contrôleurs (ces attaques sont appelées attaques *peer-to-peer*). Ces dernières attaques sont particulièrement difficiles à détecter dans la mesure où elles ne nécessitent aucun transfert de données dans la mémoire principale. Cependant, ce type d'attaque n'est pas réalisable sur tous les *chipsets*[22] actuels [LSLND10]. Les accès DMA peuvent être divisés en deux catégories, selon que le processeur intervient ou pas dans la réalisation de l'accès. Si l'accès est initié par le périphérique lui-même, on parle de périphérique disposant d'un accès *bus master*. Ces périphériques, par exemple les périphériques *FireWire*, peuvent alors prendre le contrôle du bus et réaliser des transferts de données sans aucune intervention du processeur. Ces attaques sont donc particulièrement difficiles à détecter. A l'opposé, si l'accès à la mémoire est contrôlé par le processeur, le logiciel malveillant qui réalise cet accès doit nécessairement s'exécuter sur le processeur, ou au moins doit posséder un programme complice s'exécutant sur le processeur.

Les attaques DMA ont montré leur efficacité pour contourner les mécanismes de protection mémoire implémentés par un système d'exploitation. Loïc Duflot a montré par exemple qu'un code malveillant avec des privilèges réduits peut initier des transferts DMA dans les contrôleurs USB UCI [Duf07] de façon à obtenir les mêmes privilèges que ceux du noyau. M. Dornseif [Dor04] a ainsi montré comment un périphérique *FireWire* peut accéder à la totalité de la mémoire par l'intermédiaire du contrôleur *FireWire* connecté sur une machine Linux, Mac ou BSD. A. Boileau [Boi06] a poursuivi ces travaux sur les systèmes Windows et a montré qu'il pouvait contourner le mécanisme d'identification de connexion grâce à un tranfert DMA depuis un périphérique *FireWire* simplement connecté à la machine. D'après D. Maynor [May05], les contrôleurs USB On-The-Go présenteraient les mêmes caractéristiques et pourraient être abusés par des périphériques malveillants tentant d'exécuter ce même type de transfert DMA. Enfin, L. Duflot [DLM09] a montré qu'en exploitant à distance une vulnérabilité dans le *firmware* d'un contrôleur réseau, il est possible de lui faire réaliser des accès DMA de façon à obtenir un accès administrateur à distance.

Les attaques visant la mémoire, qu'elles soient menées depuis le processeur ou depuis des contrôleurs d'entrées/sorties malveillants, peuvent être classifiées, selon leur cible : les composants logiciels (applications, système d'exploitation, hyperviseur), le processeur lui-même et son environnement d'exécution (par exemple ses registres) ou même les mémoires ou registres internes des contrôleurs d'entrées/sorties. Nous présentons brièvement chacune de ces classes dans les sous-sections suivantes.

[22] Le chipset est un ensemble de puces électroniques chargé d'interconnecter les processeurs à d'autres composants matériels tels que les mémoires, les cartes graphiques, les cartes réseau, les contrôleurs de disque, etc.

2.1.2.3 Attaques ciblant les applications

Un attaquant peut corrompre la mémoire d'une application particulière s'exécutant sur la machine cible. Cibler la mémoire d'une seule application peut avoir l'avantage d'être relativement difficile à détecter et éradiquer. De telles attaques sont en général le résultat d'un débordement de tampon. Dans le contexte avionique, on peut par exemple penser à une tâche de faible criticité qui veut provoquer la défaillance d'une application critique s'exécutant sur la même plateforme, à l'aide de ce type de corruption mémoire. Elle peut aussi modifier le comportement de l'application critique, pour la faire entrer dans une boucle infinie par exemple. De telles attaques peuvent permettre aussi de contourner des mécanismes de protection comme des systèmes d'authentification : l'application non critique peut essayer de désactiver les contrôles réalisés par l'application critique en modifiant son espace mémoire.

2.1.2.4 Attaques ciblant le système d'exploitation

Les attaques en mémoire peuvent cibler l'espace mémoire du noyau du système d'exploitation. Corrompre un noyau de système d'exploitation est bien sûr particulièrement intéressant pour un attaquant puisque cela signifie potentiellement corrompre toutes les applications s'exécutant sur ce noyau. Certains maliciels se spécialisent dans ce type de corruption. Ce sont les *rootkits noyau*. Ils tentent d'exploiter des vulnérabilités dans le code des noyaux de systèmes d'exploitation de façon à pouvoir y injecter du code malveillant. La corruption d'un noyau ayant de très graves conséquences, il est très important de le protéger efficacement. Cependant, la complexité des noyaux de système d'aujourd'hui fait que cette tâche est particulièrement ardue. De plus, les mécanismes de protection du noyau sont en général implémentés dans le noyau lui-même (qui correspond souvent au mode le plus privilégié du processeur), et peuvent donc être rapidement desactivés par un attaquant ayant réussi à injecter du code dans le noyau. Les attaques visant les noyaux de système sont donc particulièrement redoutables, quels que soient le contexte et le domaine d'application.

2.1.2.5 Attaques ciblant l'hyperviseur

Les machines virtuelles sont de plus en plus utilisées aujourd'hui, en particulier depuis l'émergence du *Cloud Computing*. Les machines virtuelles sont gérées par un gestionnaire de machines virtuelles (ou hyperviseur). Si la virtualisation a des atouts certains, elle peut aussi être la source potentielle de nouveaux problèmes de sécurité. Différents types d'attaques ont déjà été recensées [xen14, vir14, vmw14]. En effet, les machines virtuelles partagent la même architecture physique, puisqu'elles s'exécutent sur la même machine, et l'allocation des ressources qu'elles utilisent est gérée et contrôlée par l'hyperviseur. Une machine virtuelle peut donc profiter de ce partage pour essayer d'accéder ou modifier des données d'une machine virtuelle s'exécutant sur la même machine physique. Plus grave, encore, une machine virtuelle peut tenter de corrompre l'hyperviseur lui-même. La corruption de l'hyperviseur peut entraîner des conséquences graves puisque c'est lui qui

assure l'isolation spatiale et temporelle des machines virtuelles. On peut donc imaginer des attaques qui pourraient essayer de rompre cette isolation ou qui pourraient corrompre l'hyperviseur de façon à modifier directement la mémoire de certaines machines virtuelles. Des vulnérabilités de ce type ont déjà été identifiées dans certains hyperviseurs [Cou13].

2.1.2.6 Attaques ciblant l'environnement du processeur

L'environnement du processeur peut également être la cible d'attaques. En particulier, il peut être intéressant de modifier les registres d'un processeur et même son micro-code. Certains registres sont très importants, par exemple le registre `idtr` sur l'architecture x86, qui contient l'adresse de la table des interruptions. Les registres `cr0`, `cr3`, `cr4` ont aussi des rôles très importants et la corruption de certains bits de ces registres peut totalement changer le fonctionnement de la machine. Loïc Duflot a par exemple montré [DLM09] qu'il est possible d'exécuter du code malveillant dans les gestionnaires des interruptions SMI[23] (gestionnaires d'interruptions qui sont exécutés dans le mode SMM[24] du processeur). Cette corruption repose sur la modification d'un registre interne particulier du processeur. L'architecture PowerPC de type RISC contient un nombre élevé de registres. Parmi ces registres, le `MSR` (*Machine Status Register*) permet de définir le mode de privilège du processeur : utilisateur (non-privilégié) ou superviseur (privilégié). Il convient donc de ne pas le laisser accessible à n'importe quel processus. D'autres registres peuvent aussi être intéressants pour un attaquant : `IVOR0-41`[25] (registres contenant les vecteurs d'interruption), `L2CSR0-1`[26] (registres contenant la configuration du cache de niveau L2) ou encore `MAS0-7`[27] (registres contenant la configuration de la MMU).

Toutes les attaques en mémoire sont bien sûr envisageables sur tout type d'architecture, qu'elles soient grand public ou avionique et il faut donc les considérer avec attention.

2.1.3 Attaques ciblant les communications

Les communications concernent tous les échanges de données entre deux ou plusieurs entités. Elles peuvent concerner des échanges d'informations entre plusieurs tâches, à l'aide de mécanismes tels que les IPC (*Inter Process Communication*) par exemple. Elles peuvent aussi représenter les échanges entre différentes machines (ou modules dans le contexte avionique) reliées par un bus de communication (tel que le classique réseau Ethernet dans le cas des PCs grand public ou un bus spécifique tel que le réseau AFDX [ari09] dans le cas des systèmes avioniques). Nous envisageons dans cette section les attaques liées à ces différentes communications. De façon générale, les attaques ciblant les communications peuvent permettre à une tâche de faible criticité de modifier les données

[23] *System Management Interrupt.*
[24] *System Management Mode.*
[25] *Interrupt Vector Offset Register* 0 à 41.
[26] *L2 Configuration Status Register* 0 et 1.
[27] *MMU Assist Register* 0 à 7.

d'une tâche plus critique via ces mécanismes de communications. Les attaques peuvent aussi être du type *man-in-the-middle*. Elles consistent alors à capturer ou modifier des données transmises entre deux entités et elles supposent que l'attaquant ait un accès physique au medium de communication.

Si l'on considère plus spécifiquement le cas d'un système embarqué avionique, trois types de communications sont en général utilisés : les communications inter-modules, les communications intra-partition et les communications inter-partitions. La figure 2.2 représente ces communications. Deux partitions sont représentées (P0 et P1), chacun contenant quatre tâches (T0 à T3).

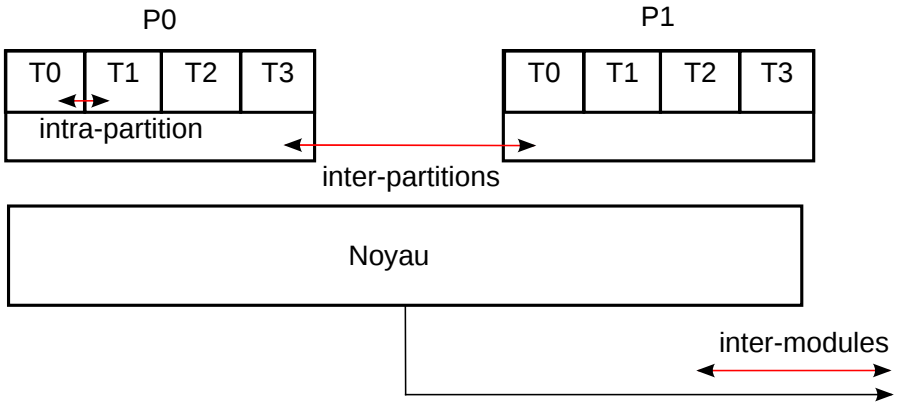

FIG. 2.2 – Les différents types de communications.

Dans l'architecture avionique de type IMA, la communication inter-modules est le plus souvent réalisée aujourd'hui en utilisant le réseau AFDX. Ce réseau est déterministe et garantit l'absence de perte de paquets. Tous les événements d'émission et réception de paquets AFDX sont préparés et configurés statiquement avant la compilation. Il n'est donc a priori pas possible de les modifier après avoir démarré les applications. En revanche, chaque partition qui émet des données, construit pendant l'exécution la *payload* correspondant à chaque émission. Une application malveillante peut donc tenter de fabriquer une *payload* d'une taille bien supérieure à la taille qu'elle indique au *driver* AFDX, de façon à provoquer un débordement de buffer dans la zone où cette *payload* est recopiée, par exemple. Par ailleurs, une partition malveillante peut aussi tenter de se faire passer pour une autre lors d'émission de paquets. Si les méta-données associées à une communication AFDX sont accessibles en écriture à chaque application, une application malveillante peut intentionnellement modifier l'identité de l'émetteur pour se faire passer pour une autre partition. Enfin, un attaquant peut également essayer d'intercepter les communications (attaque de type *man-in-the-middle*) afin de capturer ou modifier les données. Cette attaque est très difficile à réaliser car elle suppose que l'attaquant ait un accès physique au bus AFDX ou qu'il puisse modifier la configuration des *switchs* AFDX, ce qui est très peu probable et qui ne constitue donc pas une hypothèse que nous considérons dans le cadre de notre étude.

La communication inter-partitions est réalisée, toujours selon l'ARINC 653, en utilisant des messages entre partitions. Un message est un bloc de données avec une longueur finie et il est envoyé de façon périodique ou apériodique. Une partition malveillante peut tenter d'envoyer des valeurs erronées dans un message bien formé dans le but de modifier le comportement de la partition qui va recevoir ces valeurs. Cette attaque est plus intéressante si la partition qui reçoit ces messages est une partition critique. Elle peut également tenter d'envoyer des données malformées, par exemple en positionnant une fausse taille de message et en espérant provoquer un débordement de tampon dans la partition qui reçoit ce message.

Enfin, la communication intra-partition est utilisée par les processus pour qu'ils communiquent entre eux, au sein d'une même partition. Un processus est une tâche exécutée dans une partition. Deux mécanismes sont décrits dans l'API ARINC 653 pour ces communications intra-partition : les *buffers* et les *blackboards*. Un *buffer* permet de créer une file de messages en ordre FIFO (*First In First Out*). Une tâche malveillante pourrait abuser de ce mécanisme en envoyant un grand nombre de messages pour saturer la mémoire avec une file de messages trop grande ou alors en envoyant des messages de taille très importante, et ainsi perturber le fonctionnement correct des autres processus de la partition. Le *blackboard* est lui utilisé pour garder les messages jusqu'à ce qu'ils soient supprimés ou réécrits par d'autres messages. Une tâche malveillante peut facilement modifier ou supprimer les messages qui n'ont pas été écrits par elle car la zone de mémoire allouée doit être accessible à toutes les tâches de la partition qui communiquent par ce moyen. Ce mécanisme ne fonctionne que si toutes les tâches se font confiance. Ce type de corruption est cependant très limité pour un attaquant car il ne peut provoquer la défaillance que de la partition dans laquelle il a inséré une tâche malveillante. L'intérêt de l'attaquant est bien sûr plutôt de corrompre une tâche d'une autre partition ou le noyau du système lui-même.

2.1.4 Attaques ciblant la gestion du temps

Pour cadencer les différentes opérations exécutées sur un calculateur, plusieurs horloges sont disponibles. On peut citer notamment dans les architectures récentes : la *Real Time Clock (RTC)*, les *Programmable Interval Timers (PIT)*, les *Time Stamp Counters (TSC)*, les *High Precision Event Timer (HPET)*. Pour satisfaire la demande des développeurs de logiciels en ce qui concerne la précision dans la mesure du temps, ces horloges sont de plus en plus performantes. Par exemple, la puce PIT défini en 1981, manipule un compteur 16 bits à une fréquence de 1.2 MHz tandis que les nouvelles versions des puces HPET manipulent un compteur de 64 bits à une fréquence de 10 MHz. Ces puces peuvent être configurées en deux modes différents : le mode *one-shot* ou le mode périodique. Dans le mode *one-shot*, la puce exécute une interruption une seule fois quand son compteur principal atteint une valeur spécifique stockée dans un registre. Dans le mode périodique, la puce exécute une interruption à des intervalles spécifiés. Un attaquant peut donc tenter de corrompre ces horloges en modifiant les valeurs de registres de configuration associés. La corruption de ces horloges peut avoir un impact direct sur l'exécution de tâches critiques qui alors ne sont plus en mesure de respecter

leurs contraintes temps-réels, ce qui peut avoir de graves conséquences. La configuration des horloges et des interruptions est généralement modifiable uniquement par une entité avec des privilèges élevés (comme le noyau) donc seule une élévation de privilèges depuis une partition utilisateur quelconque peut donner la possibilité de corrompre les registres des horloges. Cependant, un autre facteur peut venir perturber les échéances temps réel d'une application. Il s'agit de la génération d'interruption. Lorsqu'une interruption est générée pendant l'exécution d'une application, le noyau prend la main et exécute la routine de traitement associée à cette interruption. Par exemple, la figure 2.3 représente la conséquence d'une interruption sur une partition.

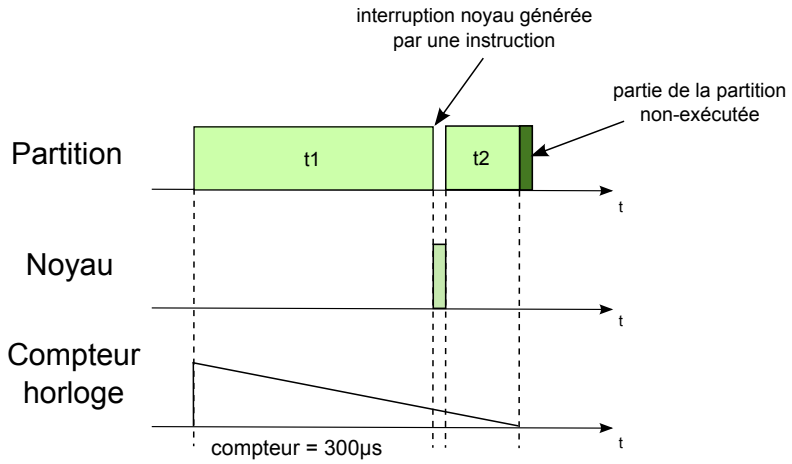

FIG. 2.3 – Exemple d'une partition de 300 microsecondes.

Dans cette figure, une partition doit s'exécuter en 300 microsecondes. Une interruption survient avant la fin et le noyau prend le relais pour la traiter. A la fin du traitement de l'interruption, la main est redonnée à la partition. Si le noyau met trop de temps à traiter l'interruption, alors la partition n'aura pas le temps de s'exécuter entièrement. Le compteur de l'horloge décrémentera jusqu'à atteindre zéro pour changer de partition, même si la partition courante n'a pas terminé son exécution. La conséquence est la suivante : une partie de la partition n'est pas exécutée (représentée par la partie foncée sur le schéma). Il est donc important d'inclure parmi les attaques qui ciblent la gestion du temps, celles qui peuvent tirer profit de la gestion des interruptions. Nous y reviendrons plus en détails dans le prochain chapitre.

2.1.5 Attaques ciblant la gestion des processus

Nous appelons "gestion des processus" la création, la modification et la suppression des processus et des *threads*. Pour créer un processus, des privilèges sont requis puisque des appels-systèmes sont nécessaires. Si un programme malveillant a la possibilité de créer des processus, il peut consommer des ressources CPU non-nécessaires menant à un déni-de-service (DoS). Une telle attaque peut être réalisée en inondant le système

d'exploitation avec des requêtes pour ralentir, voire stopper le système (en créant un nombre important de processus à l'aide d'une *fork bomb* [28] par exemple). Générer un *deadlock* est aussi un bon moyen pour un attaquant de causer un DoS. Un programme malveillant peut abuser les mécanismes de synchronisation des processus pour provoquer un blocage (*deadlock*) en monopolisant une ressource (typiquement, gardée par un sémaphore) requise par un processus légitime.

2.1.6 Attaques ciblant l'ordonnancement

Les systèmes d'exploitation modernes ont la capacité d'exécuter des processus multiples en même temps sur le même processeur. Ces systèmes d'exploitation sont appelés "systèmes d'exploitation multi-tâches". L'ordonnanceur qui organise l'exécution des processus en fonctionnement décide quand le processeur doit exécuter le processus et pour combien de temps. Quand le slot de temps dédié à l'exécution d'un processus expire, l'ordonnanceur donne automatiquement la main au prochain processus. Typiquement, pendant le démarrage du système, le noyau met en place une horloge pour activer périodiquement l'ordonnanceur. La gestion efficace des ressources par l'ordonnanceur est cruciale dans ce contexte. Si un processus est interrompu alors qu'il utilise une ressource, l'ordonnanceur ne doit pas donner la main à un autre processus qui pourrait utiliser la même ressource. Cette situation doit être évitée pour empêcher les *deadlocks*. Dans ce cadre, un attaquant peut tenter de corrompre l'ordonnancement en modifiant la configuration de l'horloge dans la zone de mémoire correspondante. Le résultat d'une telle attaque serait catastrophique car les contraintes temps-réel ne seraient alors pas respectées.

2.1.7 Attaques ciblant les mécanismes cryptographiques

Les mécanismes cryptographiques peuvent aussi être vulnérables à des attaques. Des algorithmes peuvent ainsi contenir des défauts découverts plusieurs années après leur développement, qui peuvent être exploités grâce à l'amélioration de la puissance de calcul disponible (par exemple, les collisions MD5 ont été découvertes en 2004 [WFLH04] alors que l'algorithme existe depuis 1991) et même si les algorithmes cryptographiques sont prouvés pour être corrects, leurs implémentations peuvent contenir des bugs ou des contournements qui pourraient être exploités par des attaquants. L'analyse par canaux-cachés est un exemple d'une telle exploitation [KJJ99]. La consommation de courant, les analyses des temps [Koc96] ou même l'écoute des composants [GST13] peuvent permettre de retrouver des clés secrètes RSA ou AES. Cependant, ces attaques ne sont réalisables qu'avec un accès physique au système et avec un matériel important pour réaliser toutes les mesures nécessaires. Par exemple, une clé RSA peut être reconstruite en utilisant seulement 27% de ces bits en mémoire [HS09]. Les attaques sur les mécanismes de cryptographie dépendent grandement de la gestion de la mémoire (voir sous-section 2.1.2) car si une "fuite" de mémoire est présente, des informations sur la clé secrète peuvent être divulguées, et ainsi compromettre la sécurité du système complet.

[28] Processus créés dans une boucle infinie, menant à l'épuisement des ressources du système.

2.1.8 Attaques ciblant les fonctions ancillaires

Les fonctions ancillaires correspondent à la gestion de l'alimentation, de l'*overclocking*, du contrôle de température, etc. Dans les processeurs x86, ces fonctions sont supportées par le SMM (*System Management Mode*). Ces dernières années, des vulnérabilités ont été découvertes dans ce mode. Un attaquant peut exploiter ces vulnérabilités pour exécuter du code arbitraire dans un mode privilégié [ESZ08, DLM09]. Le mode SMM est un mode 16 bits utilisé pour le contrôle de la carte mère et la gestion de l'alimentation. Seule une SMI (*System Management Interrupt*) peut faire entrer le processeur dans le mode SMM, ces SMIs sont générés par le chipset. Mais n'importe quel contrôleur avec les privilèges d'entrées/sorties peut générer ces SMIs. A partir de cette zone SMM, il est possible d'obtenir des privilèges élevés, d'utiliser un *rootkit* ou encore de contourner des restrictions de démarrage sûrs (*late launch restrictions*) [DLM09].

2.1.9 Conclusion

Nous avons, dans cette section, essayé de lister exhaustivement les attaques pouvant cibler les fonctionnalités de base d'un calculateur avionique. Comme nous l'avons explicité au début de ce chapitre, cette classification concernant les fonctionnalités de base est tirée en grande partie des attaques que l'on peut rencontrer sur les systèmes informatiques "grand public". La section suivante concerne des attaques ciblant des mécanismes spécifiques qui sont présents dans la plupart des calculateurs embarqués exécutant des systèmes critiques, pour des raisons de sûreté de fonctionnement : les mécanismes de tolérance aux fautes.

2.2 Les attaques ciblant les mécanismes de tolérance aux fautes

Les systèmes avioniques, comme bon nombre de systèmes embarqués critiques, intègrent des mécanismes de tolérance aux fautes pour être capables de continuer à fonctionner correctement même si des défaillances surviennent. La figure 2.4 représente ces différents mécanismes de tolérance aux fautes. Ces principes font références aux concepts de bases présentés dans [ALRL04].

La chaîne fondamentale des entraves à la sûreté de fonctionnement est la suivante :

$$... \Rightarrow \text{Faute} \Rightarrow \text{Erreur} \Rightarrow \text{Défaillance} \Rightarrow ...$$

La tolérance aux fautes vise à éviter l'occurence de défaillances. Elle est mise en œuvre par la détection des erreurs et le rétablissement du système. Le rétablissement du système est lui-même mis en œuvre par le traitement d'erreurs et le traitement de fautes. Le traitement d'erreurs a pour but l'élimination des erreurs de l'état du système avant l'occurrence de défaillance. Le traitement de fautes a pour but d'empêcher une

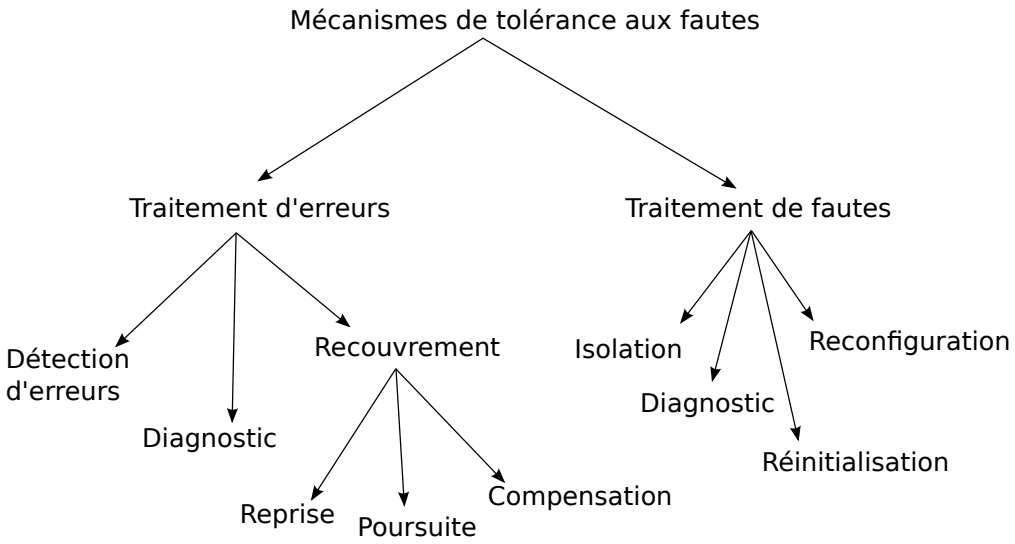

FIG. 2.4 – Schéma des différents traitements de la tolérance aux fautes.

nouvelle activation de faute dans le système. Le traitement d'erreurs est en général la première étape du rétablissement du système, le traitement des fautes en est la seconde.

Le traitement d'erreurs peut être mis en œuvre par :

- la détection d'erreur, qui permet à un état erroné d'être identifié comme tel ;
- le diagnostic d'erreur, qui permet l'évaluation des dommages par une erreur détectée ou par des erreurs propagées avant la détection ;
- le recouvrement d'erreur, où un état sans erreur est substitué à un état erroné.

Les techniques de détections d'erreur peuvent être classifiées en détections concomitantes, si elles sont réalisées pendant l'exécution du service normal, et en détections préemptives, si elles sont effectuées lors de la suspension du service (par exemple, en phase d'initialisation). La détection préemptive vise à révéler l'éventuelle présence d'erreurs latentes ou des fautes dormantes.

Concernant le recouvrement d'erreurs, trois techniques sont classiquement utilisées :

- la reprise, où la transformation de l'état erroné consiste à ramener le système à un état antérieur sain avant l'occurrence d'erreur ; ceci implique la sauvegarde de points de reprise, qui sont des sauvegardes régulières dans le temps, de l'état du système, dans lequel le système peut ensuite être restauré ;
- la poursuite, où la transformation de l'état erroné consiste à trouver un nouvel état dans lequel le système peut continuer à opérer (éventuellement en mode dégradé) ;
- la compensation, où l'état erroné contient suffisamment de redondances pour permettre sa transformation en un état sans erreur.

La reprise est en général inapplicable à un système avionique puisqu'il est impossible de ramener le système complet (incluant notamment certaines données réelles comme la position ou la vitesse de l'appareil, le niveau de carburant disponible, etc.) dans un

état sain précédent. En revanche, les systèmes avioniques appliquent en général une combinaison de techniques de poursuite et de compensation.

La seconde étape de rétablissement de système est le traitement de fautes. Il vise à empêcher de nouvelles activations de fautes. Le traitement de fautes comprend plusieurs étapes : le diagnostic, l'isolation, la reconfiguration et la réinitialisation. Le diagnostic identifie et enregistre les causes des erreurs de façon à en préciser le type et la localisation précise. L'isolation consiste à exclure de façon logicielle ou matérielle, les composants fautifs de la délivrance du service (en cela, cette étape rend les fautes dormantes). La reconfiguration peut soit provoquer l'exécution du service sur un composant supplémentaire à disposition, soit répartir les tâches sur l'ensemble des composants disponibles non fautifs. La réinitialisation vérifie et met à jour la nouvelle configuration, et enregistre cette nouvelle configuration dans des tables du système.

Tous ces mécanismes de tolérance aux fautes peuvent être des cibles pour un attaquant. Par exemple, si l'attaquant est capable d'empêcher la détection d'erreurs, il peut en conséquence empêcher le recouvrement du système en cas d'erreur et ainsi mener le système à défaillance. S'il peut corrompre le diagnostic d'erreur, il peut également empêcher la reconfiguration du système. La corruption d'un recouvrement d'erreur peut provoquer la substitution d'un état d'erreur bien choisi à l'état courant. De la même façon, le diagnostic de fautes peut être compromis pour identifier un composant correct comme défaillant ou un composant fautif comme correct. Une passivation de fautes corrompue peut désactiver un composant correct à la place du composant défaillant, et une reconfiguration défaillante peut modifier la structure du système afin que les tâches critiques ne soient plus exécutées.

Dans les sections suivantes, nous envisageons plus en détail les attaques pouvant cibler le traitement des erreurs (en particulier, les étapes de détection des erreurs et recouvrement des erreurs) et le traitement des fautes.

2.2.1 Détection d'erreurs

Du point de vue d'un attaquant, les mécanismes de détection d'erreur peuvent être une cible intéressante. En effet, si un attaquant est capable d'empêcher la détection d'erreur, alors aucun recouvrement ne peut être effectué, ce qui peut causer une défaillance du système. Inversement, si un attaquant provoque la détection d'erreur trop fréquemment, il peut déclencher une procédure de recouvrement incorrecte, ce qui peut provoquer l'épuisement des unités de rechange et ainsi rendre inefficace les mécanismes de tolérance aux fautes.

Les moyens utilisés pour la détection d'erreurs peuvent varier en fonction du niveau où la vérification est effectuée et le compromis coût/efficacité sélectionné. Les formes les plus populaires sont les suivantes :

- Les codes détecteur d'erreur ;
- La duplication et comparaison ;
- Les contrôles temporels et d'exécution ;
- Les contrôles de vraisemblance ;
- Les contrôles de données structurées.

Nous décrivons dans la suite trois d'entre elles : les contrôles de vraisemblance, la duplication et comparaison et les contrôles temporels et d'exécution car ils sont largement utilisés dans l'industrie.

2.2.1.1 Contrôle de vraisemblance

L'implémentation de mécanismes de contrôle de vraisemblance nécessite peu de coûts additionnels, en comparaison au coût des éléments fonctionnels du système. Beaucoup de contrôles peuvent être implémentés pour détecter si les erreurs surviennent depuis une large sélection de fautes, mais leur couverture est généralement limitée. Ce mécanisme peut être utilisé par exemple pour détecter les erreurs de valeurs, les violations des mécanismes de protection de la mémoire ou pour vérifier la conformité des entrées/sorties. Ces contrôles sont relativement faciles à faire échouer sous réserve de connaître la façon dont le système se comporte avec des données non conformes. Envoyer des données aléatoires (technique de *fuzzing*) permet de tester si les contrôles de vraisemblance peuvent résister à tout type de données d'entrée. Si des erreurs ne sont pas détectées alors le système peut continuer son exécution en engendrant de possibles défaillances.

2.2.1.2 Duplication et comparaison

La duplication et comparaison est un mécanisme de détection d'erreurs largement utilisé, malgré un coût matériel important, car il est simple à mettre en place. La mise en œuvre repose sur l'utilisation d'au moins deux unités redondantes qui sont indépendantes du point de vue de la création et de l'activation des fautes : il est nécessaire de s'assurer que soit les fautes sont crées ou activées indépendamment dans les unités redondantes, soit que, si une même faute provoque des erreurs dans chaque unité, ces erreurs sont différentes. Si on considère les fautes physiques internes, les unités peuvent être identiques dans la mesure où l'on peut supposer que les compsants matériels défaillent de manière différente dans chaque unité. Dans le cas de fautes physiques externes, il faut éviter des fautes de mode commun en isolant physiquement les deux unités ou décalant dans le temps le traitement sur les deux unités.

Face à la duplication et comparaison, un attaquant peut tenter de corrompre chaque unité de façon à ce que la détection soit inefficace. L'éloigement géographiquement des unités peut résoudre en partie le problème lorsqu'il s'agit d'attaques physiques mais si l'on considère les attaques logicielles (et donc les fautes de conception du logiciel), il est nécessaire d'avoir recours à la diversification des logiciels et de matériels. En effet, des logiciels et matériels de conception et de mise en œuvre différentes ne sont pas vulnérables aux mêmes attaques par construction.

2.2.1.3 Contrôles temporels et d'exécution

Dû à ses coûts très limités, les contrôles temporels par des "chiens de garde" (*watchdog timers*) sont les plus largement utilisés pour la détection d'erreurs durant l'exécution. Un *watchdog* peut être utilisé dans différentes situations telles que la détection de la

défaillance d'un périphérique en contrôlant son temps de réponse (suite à l'envoi d'un signal) qui ne doit pas dépasser une valeur maximale ou la surveillance de l'activité des processeurs. Dans ce dernier cas, le *watchdog* doit être périodiquement rafraîchi par le processeur. Ainsi, si le comportement du processeur est altéré de sorte que le *watchdog* n'est plus rafraîchi avant qu'il n'expire, une exception sera levée. Une telle approche peut être utilisée, pour sortir d'une situation de blocage ou d'une boucle infinie, ou pour détecter l'arrêt d'un processeur distant.

Une attaque simple sur cet élément consiste à empêcher le signal de revenir à l'émetteur et donc de faire passer un programme ou un périphérique comme défaillant alors qu'il ne l'est pas. Cette technique peut être mise en œuvre par un déni-de-service du programme visé (par exemple, par saturation de requêtes d'appels-systèmes) ce qui engendrera une augmentation du temps processeur ralentissant ainsi fortement le traitement des requêtes et donc le temps de réponse associé.

2.2.2 Recouvrement d'erreurs

En concordance avec la détection d'erreurs, le recouvrement d'erreurs est utilisé pour continuer à délivrer le service du système. Trois types de recouvrements peuvent être effectués : la reprise, la poursuite et la compensation. La reprise est le moyen le plus utilisé ; il consiste en la sauvegarde régulière de l'état du système pour permettre de le restaurer si une erreur est détectée. Le système est sauvegardé avec des points de restauration, lesquels peuvent être possiblement corrompus par un attaquant. La corruption de tout ou partie d'un seul point de restauration suffit à mettre en danger le système. Il est donc important de vérifier les privilèges nécessaires à la création et la modification des régions de mémoire dans lesquelles ces points de restauration sont créés.

La poursuite constitue une alternative ou une approche complémentaire à la reprise. La poursuite consiste à chercher un nouvel état acceptable pour le système, à partir duquel il sera possible de poursuivre l'exécution du système (éventuellement dans un mode dégradé). Dans le cas où la poursuite est la technique de tolérance aux fautes adoptée par le système, l'attaquant peut faire en sorte de forcer l'utilisation du mode dégradé, en provoquant le plus souvent possible des erreurs.

La compensation d'erreur nécessite que l'état du système comporte suffisamment de redondance pour permettre en dépit des erreurs pouvant l'affecter, sa transformation en un état exempt d'erreur. Un exemple typique est celui des architectures à base de vote majoritaire où la présence d'erreurs n'aura aucune conséquence pour le système. Un exemple typique de la détection et compensation d'erreur est l'utilisation des composants autotestables exécutant en redondance active le même traitement ; 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, dans ce cas, se limite à une commutation éventuelle de composants. C'est sur ce principe que fonctionne le système de gestion de commandes de vol des Airbus 320/330/340. Lorque la compensation d'erreurs est utilisée, et donc de la redondance, un attaquant peut faire en sorte de provoquer des erreurs dans chacune des unités redondantes, de façon à faire en sorte que la compensation ne soit plus possible

puisqu'il ne subsiste plus d'unité considérée correcte dans le système.

2.2.3 Traitement de fautes

Le traitement de fautes a pour but d'empêcher les fautes d'être activées une nouvelle fois. Le traitement de fautes consiste en : diagnostic, isolation, reconfiguration et réinitialisation. Toutes ces techniques peuvent être la cible d'attaques. Par exemple, des attaques peuvent faire en sorte de faire échouer le diagnostic de façon à diagnostiquer à tort un composant non-défectueux comme défectueux. Un attaquant peut de même tenter de provoquer l'isolation d'un composant non-défectueux ou provoquer la poursuite du système avec un composant défectueux. Il peut également provoquer à tort la mise à jour d'une nouvelle configuration du système. Toutes ces attaques peuvent amener le système à défaillance. En effet, le diagnostic à tort d'un composant non-défectueux comme défectueux permet d'éliminer des composants valides du système et donc à terme de forcer le système à passer dans un mode dégradé. De même, si l'attaquant parvient à provoquer l'isolation de tous les composants non défectueux, alors il peut provoquer une exécution totalement erronée du système. Enfin, l'attaquant peut également cibler le mécanisme de mise à jour de la configuration. S'il parvient à mettre à jour la configuration de façon à activer une nouvelle configuration erronée, il peut amener le système à défaillir.

2.2.3.1 Isolation de fautes

Pour isoler une faute, le composant concerné est supprimé mais dans un scénario malveillant, l'attaquant peut essayer de provoquer volontairement une faute pour isoler un composant sain. Il peut ainsi provoquer la passivation de toutes les unités du système et provoquer son arrêt. L'attaquant peut aussi directement cibler le mécanisme d'isolation de fautes de façon à ne pas isoler un composant lorsqu'il est défectueux. Ces attaques peuvent mener le système à se comporter d'une façon totalement arbitraire et ainsi provoquer sa défaillance. Une autre attaque possible consiste à modifier le mécanisme d'isolation de fautes pour faire en sorte que le mécanisme détecte tous les composants comme défectueux (ce qui aura comme conséquence de forcer l'isolation pour tous les composants) ou de faire en sorte que ce mécanisme ne détecte aucun composant défectueux (ce qui aura pour conséquence de laisser les défaillances se produire suite à un état incohérent du système).

2.2.3.2 Reconfiguration

La reconfiguration consiste à modifier la configuration du système (éventuellement dans un mode dégradé) pour le réactiver dans un état sans faute. Un attaquant peut tirer avantage de cette reconfiguration en modifiant le processus qui effectue la reconfiguration pour établir sa propre reconfiguration. Une reconfiguration peut impliquer l'abandon de certaines tâches ou la ré-allocation de certaines tâches aux composants restants pour délivrer un service acceptable. Cependant, le service délivré après reconfiguration peut

être en mode dégradé, si la reconfiguration est compromise. La configuration peut être modifiée si par exemple, les accès en écriture à la mémoire où est stockée cette configuration sont trop permissifs. L'attaquant peut alors provoquer l'arrêt de toutes les tâches ou empêcher la ré-allocation de certaines tâches.

2.3 Contexte et hypothèses d'attaques

Dans la section précédente, nous avons classifié les différentes attaques possibles en deux catégories : mécanismes de base et mécanismes de tolérance aux fautes. Dans cette section, nous proposons d'exposer l'ensemble des hypothèses et contraintes qui caractérisent l'environnement dans lequel nous effectuons nos expérimentations d'attaques. En particulier, nous allons exposer les hypothèses d'attaques mais aussi les conditions particulières dans lesquelles nous allons étudier la faisabilité de ces attaques. Un système critique avionique répond à un certain nombre de contraintes particulières et nous devons en tenir compte pour établir les hypothèses. Dans la première sous-section, nous détaillons les types d'attaquants et les hypothèses d'attaques que nous considérons dans cette étude. Dans la sous-section suivante, nous détaillons le type d'application malveillante que nous avons utilisé pour cette étude, sa configuration et la façon dont nous l'avons modifiée.

2.3.1 Attaquants

Notre étude se focalise sur les attaques logicielles. En effet, nous considérons que des personnes malveillantes n'ont pas d'accès physique au système. Si un tel accès physique est donné, il est plus simple, pour provoquer des défaillances, de débrancher des cartes ou des câbles plutôt qu'avoir recours à des applications malveillantes. Par exemple, le débranchement des câbles AFDX interrompt les communications entre modules, ce qui provoque des erreurs à répétition pour finalement probablement mettre en fonctionnement les unités redondantes si elles existent ou simplement arrêter les modules qui ne peuvent plus communiquer.

Nous considérons donc les trois hypothèses suivantes :

- L'attaquant n'a pas d'accès physique au système en exécution ;
- Les pilotes et l'équipage sont considérés comme des personnes de confiance, ainsi que tout le personnel de maintenance ;
- Les passagers, les utilisateurs de réseaux publics et les développeurs COTS sont considérés comme des personnes potentiellement malveillantes ;

Nous donnons dans la suite quelques informations sur cette classification des personnes potentiellement malveillantes ou non ainsi que sur leur liberté d'actions s'ils sont malveillants. Précisons que ces hypothèses d'attaques peuvent sembler peu probables, mais, dans le cas des analyses de sécurité et en accord avec Airbus, nous avons volontairement opté pour des hypothèses relativement pessimistes.

Les passagers sont considérés comme potentiellement malveillants. En effet, les passagers ont la possibilité d'embarquer à bord de l'avion leur ordinateur portable, leur

tablette, leur smartphone, etc. Ils peuvent ainsi, à l'aide de ces matériels connectés, tenter de se connecter au réseau de l'avion et tenter ensuite de progresser dans ce réseau de façon à accroître leurs privilèges et ainsi essayer d'atteindre des calculateurs connectés de façon plus ou moins directe au système de vol.

Par ailleurs, les passagers, même bienveillants, peuvent posséder des logiciels infectés sur leur machine personnelle, qui peuvent réaliser ce type d'attaque à leur insu. Il est même possible que cette injection survienne directement lorsque le passager est dans l'avion. En effet, juste avant la phase de décollage, lorsque l'accès au réseau Internet est encore autorisé pour les passagers, on peut imaginer qu'un internaute malveillant puisse exploiter une vulnérabilité d'un PC portable ou d'une tablette d'un passager, qu'il puisse installer à cet instant précis un logiciel malveillant grâce à cette vulnérabilité et qu'il puisse tenter ensuite de mener une attaque à partir de cet équipement corrompu. Enfin, on peut même imaginer ce type d'attaque pendant le vol, puisque certaines compagnies autorisent d'ores et déjà l'utilisation du réseau Internet en vol.

Certaines personnes ont des accès privilégiés à l'avion en dehors des périodes de vol pour la maintenance de l'avion par exemple, le ravitaillement, mais aussi pour l'installation d'équipements dans l'appareil, notamment dans la cabine. Ces opérateurs sont considérés comme des personnes de confiance et ne sont donc pas susceptibles d'installer volontairement des applications malveillantes lors de leur accès privilégié à l'avion. Cependant, il est envisagé dans un futur proche d'équiper les opérateurs de maintenance de PC portables personnels qui seront utilisés pour effectuer cette maintenance (qui aujourd'hui est réalisée avec du matériel spécifique). Si ce PC portable peut être par ailleurs utilisé en dehors des phases de maintenance de l'avion, pour un usage personnel (connexion au réseau Internet par exemple), il est alors tout à fait possible qu'il puisse être infecté par un logiciel malveillant. Lorsque ce PC est ensuite connecté au réseau de l'avion lors de la phase de maintenance, le logiciel malveillant peut tenter d'exploiter une vulnérabilité, pour ensuite tenter de se propager sur ce réseau. Même si aujourd'hui, ce type de menace n'est pas envisageable, il peut le devenir dans un futur proche (Airbus a d'ailleurs d'ores et déjà lancé des études sur le sujet pour y faire face [Laa09]).

Comme nous l'avons indiqué dans le premier chapitre, l'introduction de logiciels COTS dans les systèmes critiques est de plus en plus envisagé. Sans envisager la malveillance des développeurs de COTS, on peut cependant supposer que ces derniers développent leurs applications en ne respectant pas de strictes règles de codage et que par conséquent, leurs logiciels sont susceptibles de contenir des vulnérabilités, qui peuvent être exploitées pour exécuter des attaques. Dans le cas le plus favorable, ces COTS sont utilisés uniquement pour l'implémentation de partitions non critiques (c'est le cas que nous supposons dans cette thèse) mais on peut également imaginer que des COTS puissent être également utilisés dans le développement de logiciels critiques. Par ailleurs, les logiciels COTS ne sont en général pas développés pour le monde embarqué (par exemple vlc[29] pour le *streaming*) et sont donc susceptibles de ne pas correspondre entièrement aux besoins des systèmes avioniques. Ces logiciels sont bien sûr testés et évalués du point de vue de la sécurité avant d'être insérés dans l'avion mais il est possible que

[29] `http://www.videolan.org/vlc/`

malgré tout, étant donné leur complexité, un certain nombre de vulnérabilités subsistent, en particulier dans les couches les plus proches du matériel, et que ces vulnérabilités puissent être exploitées. On peut aussi envisager l'hypothèse qu'un logiciel ou matériel COTS puisse contenir intentionnellement des fonctions malveillantes (une porte dérobée par exemple).

2.3.2 Hypothèse d'attaque : une application non critique malveillante

Dans le cadre des expérimentations d'attaques que nous avons menées dans cette thèse et qui font l'objet du chapitre suivant, nous avons fait le choix d'utiliser une application malveillante compatible ARINC 653 déjà installée parmi les autres applications sur un calculateur qui exécute un noyau temps-réel critique. Cette application est non critique et l'objectif est de vérifier si elle peut avoir un impact sur la sécurité d'une application critique ou du noyau système lui-même. La façon dont a pu être installé cette application malveillante n'est pas l'objet de cette thèse mais elle peut faire intervenir les personnes potentiellement malveillantes que nous avons présentées dans la section précédente. L'installation de ce type d'application malveillante, même non critique, est bien sûr très improbable aujourd'hui, compte tenu des mécanismes de protection (physiques et logiques) déjà mis en place dans les avions. Cependant, dans le cadre de cette étude, nous nous plaçons volontairement dans un cadre très pessimiste, où nous supposons qu'elle a pu être installée, et nous nous focalisons sur l'étude des effets potentiels de cette application, afin de mieux s'en protéger.

Dans le cadre de nos expérimentations, nous disposons d'un exécutif temps réel expérimental, en cours de développement, fourni par Airbus. La configuration du noyau et des applications est réalisée avant la compilation grâce à des fichiers textes qui permettent de générer le code approprié pour le noyau. Cette configuration comprend majoritairement les accès à la mémoire pour chaque partition et le noyau mais également les durées de chaque partition et la durée totale d'un cycle (l'exécution séquentielle de toutes les partitions), les communications inter-partitions si elles existent, le comportement du *Health Monitor* (HM) pour chaque partition, les appels-système, etc.

Pour créer notre application malveillante, nous avons copié le code source en langage C d'une application avionique basique. Nous avons ensuite modifié l'application en langage assembleur pour ne pas dépendre des limitations du langage C par rapport à un langage de bas niveau. Effectivement, un langage de bas niveau peut directement avoir accès à la mémoire et à tous les registres (du mode utilisateur).

En ce qui concerne le fonctionnement de l'application malveillante, elle peut réaliser toutes les actions disponibles depuis un mode utilisateur, qui peuvent éventuellement provoquer des erreurs, notamment :

- exécuter des instructions privilégiées ;
- accéder à des registres spéciaux ;
- accéder à des données dans une zone mémoire interdite ;
- exécuter des instructions dans une zone mémoire interdite.

Le but est d'observer le comportement du noyau lorsque ces tests sont effectués. Si le mécanisme de gestion des erreurs est correctement réalisé alors aucune conséquence ne

sera constatée. Comme nous le verrons dans le prochain chapitre, nous avons dû instrumenter le noyau pour observer son comportement et celui des applications, notamment pour mesurer les durées d'exécution ainsi que pour contrôler les erreurs provoquées.

Conclusion

Nous avons détaillé dans ce chapitre une classification de différentes attaques possibles sur les systèmes embarqués, qu'elles ciblent les mécanismes de base (processeur, gestion de la mémoire, communications, gestion du temps, gestion des processus, ordonnancement, cryptographie, fonctions ancillaires) ou qu'elles ciblent les mécanismes de la tolérance aux fautes (mécanismes de traitement d'erreur et mécanismes de traitement de fautes). Nous avons également précisé le contexte de nos expérimentations en listant les hypothèses d'attaques ainsi que le contexte d'exécution de ces attaques. Nous avons ainsi indiqué comment l'application malveillante, que nous utilisons pour lancer des attaques, a été configurée et installée parmi les autres applications.

Dans le chapitre suivant, nous décrivons la plateforme d'expérimentation utilisée pour la réalisation de nos attaques, le noyau expérimental, les partitions ainsi que l'environnement de développement. Nous développerons également différentes attaques que nous avons pu mettre en œuvre.

Chapitre 3

Mise en œuvre de la méthodologie sur une plateforme IMA expérimentale

Introduction

Ce chapitre décrit 1) la méthodologie employée pour mettre en œuvre notre plateforme IMA d'expérimentations ainsi que 2) nos expérimentations et leurs résultats. Cette plateforme est composée d'un calculateur, d'une sonde JTAG, de l'IDE Codewarrior et d'un noyau temps-réel expérimental conçu par Airbus. Le noyau est développé selon le standard avionique ARINC-653 (*"Avionics Application Software Standard Interface"*). Ce standard est une interface pour le partitionnement temporel et spatial des ressources des systèmes embarqués. Les éléments constituant cette plateforme sont détaillés dans la suite de ce chapitre.

Dans la section 3.1, nous décrivons le noyau temps-réel expérimental d'Airbus, son fonctionnement et sa configuration. Ensuite, nous détaillons l'architecture de la plateforme matérielle que nous avons utilisée, le P4080, et les fonctionnalités utilisées par le noyau (section 3.2). Dans la section 3.3, nous présentons les composants de la plateforme IMA utilisés. Puis, nous présentons les attaques (ainsi que leurs résultats) ciblant les mécanismes de base, notamment en ce qui concerne la gestion de la mémoire (section 3.4), les communications (section 3.5) et la gestion du temps (section 3.6). Enfin, nous décrivons les attaques (ainsi que leurs résultats) ciblant les mécanismes de tolérance aux fautes (section 3.7). Ces différents travaux ont fait l'objet de publications dans [DADN13] et [DDAN13].

3.1 Système expérimental d'Airbus : noyau et partitions

Le noyau temps-réel expérimental d'Airbus permet d'exécuter des applications en respectant le principe de partitionnement (ARINC 653). Le partitionnement spatial est

opéré par la MMU et le partitionnement temporel par le MPIC. Les partitions sont exécutées dans l'ordre prévu par la configuration des fichiers comme vu précédemment. Le noyau s'initialise en réalisant les opérations suivantes :

- l'activation des permissions de la zone de mémoire (MMU) ;
- l'activation du compteur de temps (*Time Base*) ;
- l'initialisation des interruptions (MPIC) ;
- l'initialisation des caches.

Lorsque l'initialisation du noyau et des partitions est terminée, la première partition est exécutée suivant l'ordre défini précédemment. Cette partition exécute son code pendant quelques microsecondes. Puis, à la fin de sa tranche de temps, la partition suivante prend le relais et ainsi de suite jusqu'à l'arrêt du système. Nous pouvons représenter de façon très simple l'ordonnancement des partitions comme suit :

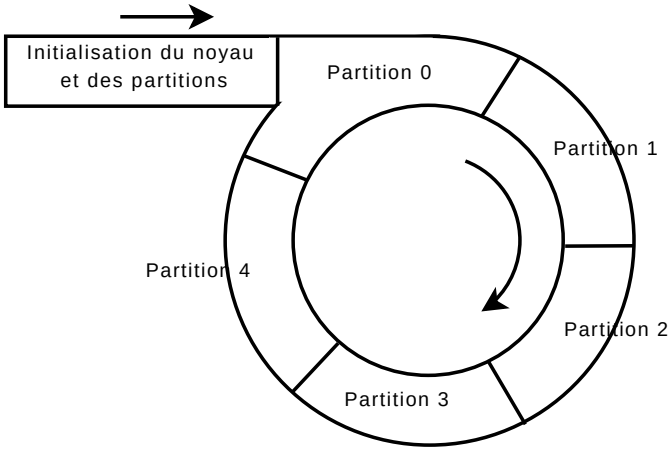

FIG. 3.1 – Exemple d'ordonnancement avec 5 partitions.

Ensuite, la première partition s'exécute et lorsque sa tranche de temps (*time slot*) est terminée, le noyau opère un changement de partition qui consiste à sauvegarder le contexte de la partition courante. Pour cela, un changement de contexte (*context switching*) est réalisé. Nous décrivons cette opération dans la sous-section 3.1.2. Le contexte d'une partition correspond au contenu de plusieurs registres :

- GPR (*General Purpose Register*) : les 32 registres généraux (de 32 bits chacun) ;
- FPR (*Floating-Point Register*) : les 32 registres à virgule flottante (de 64 bits chacun) ;
- PC (*Program Counter*) : l'adresse de la prochaine instruction (32 bits) ;
- MSR (*Machine State Register*) : l'état du processeur (dont les privilèges) (32 bits) ;
- LR (*Link Register*) : l'adresse de destination d'une branche[30] (32 bits) ;

[30] Á noter qu'en langage PowerPC, une "branche" ou un "branchement" correspond à un saut d'une instruction à une autre.

- CTR (*Count Register*) : ce compteur est décrémenté et testé pendant l'exécution des instructions de branchement (32 bits) ;
- XER (*Integer Exception Register*) : ce registre permet de connaître des informations sur les opérations d'entiers (retenu, débordement) ainsi que le nombre d'octets transmits par une instruction sur les chaînes de caractères (32 bits) ;
- CR (*Condition Register*) : ce registre contient 8 registres de 4 bits permettant d'utiliser jusqu'à 8 comparaisons d'entiers simultanées (32 bits) ;
- FPSCR (*Floating-Point Status and Control Register*) : ce registre contient des informations d'erreurs sur les nombres à virgules flottantes (64 bits).

Un contexte de partition contient donc 416 octets qui sont sauvegardés dans la mémoire utilisateur de la partition (accessible en lecture/écriture) à chaque fin d'exécution de la tranche de temps de la partition puis restaurés à chaque début de tranche de temps. Á noter que le registre MSR qui permet, en autre, de passer du mode superviseur au mode utilisateur n'est pas restauré depuis la mémoire comme les autres mais défini statiquement (afin d'interdire à la partition ayant un accès sur cette zone de modifier ses privilèges lors de la prochaine restauration de contexte).

3.1.1 Hypothèses concernant la partition malveillante

Comme pour les autres applications, les règles de partitionnement spatial et temporel sont appliquées. La partition non critique depuis laquelle nous allons mener nos attaques s'exécute donc périodiquement comme toutes les autres applications, pendant 1.5 millisecondes sur les 5 millisecondes disponibles (durée correspondant à un cycle d'exécution de toutes les partitions). Elle est exécutée en dernière position (7ème position). Pour le partitionnement spatial, nous avons configuré la partition pour utiliser une zone de 1Mo de données en lecture/écriture et 1Mo d'instructions en lecture/exécution et aucun autre accès à la mémoire. Ces zones ne sont pas en conflit avec d'autres zones (les zones ne se chevauchent pas) pour éviter de donner des accès à d'autres applications.

Pour notre application non critique malveillante, nous utilisons les mêmes privilèges que les autres applications, à savoir un mode utilisateur qui est restreint sur certaines instructions privilégiées (contrairement au mode superviseur). En ce qui concerne la protection mémoire, elle est assurée par la MMU (*Memory Management Unit*). Ce composant est situé dans le processeur pour contrôler les accès à la mémoire. Une entrée dans la MMU correspond à une entrée dans une TLB (*Translation Lookaside Buffer*) qui est une table contenant tous les accès aux zones mémoires configurées.

Deux entrées sont créées au minimum, une pour les données et une autre pour les instructions. D'autres entrées sont également utilisées pour les accès du noyau. Comme pour les autres applications, l'application malveillante ne peut accéder ni aux périphériques ni au noyau en dehors des mécanismes prévus tel que les appels système. La figure 3.2 représente un exemple d'entrées dans la TLB lorsque les partitions 5 et 6 sont exécutées.

Dans cet exemple, la partition 6 a un accès de l'adresse 0x0070_0000 à 0x0080_0000 en lecture et écriture pour la modification des données et un autre accès de 0x0080_0000 à 0x0090_0000 en lecture et exécution pour exécuter les instructions. La partition 5 a un

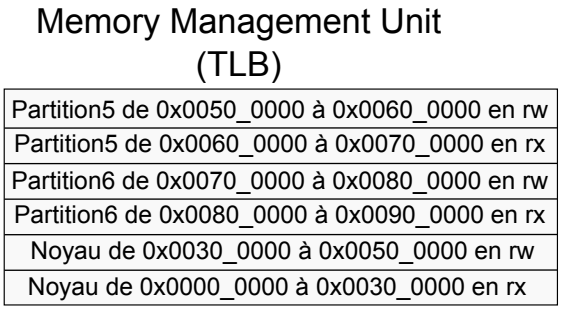

FIG. 3.2 – Exemple d'entrées dans la TLB avec deux partitions et un noyau.

accès de 0x0050_0000 à 0x0060_0000 en lecture et écriture pour les données et un autre accès de 0x0060_0000 à 0x0070_0000 en lecture et exécution pour les instructions. Le noyau a lui aussi un accès pour ses données de 0x0030_0000 à 0x0050_0000 en lecture et écriture et de 0x0000_0000 à 0x0030_0000 en lecture et exécution pour ses instructions. Les accès entre le noyau et les applications peuvent paraître similaires dans la TLB mais le noyau s'exécute lui en mode superviseur alors que les applications s'exécutent en mode utilisateur. Si un accès non autorisé est effectué tel qu'un accès en dehors des limites prévues par ces entrées dans la TLB alors une réaction adaptée doit être mise en œuvre, comme par exemple l'arrêt ou le redémarrage de la partition.

3.1.2 Phases d'exécution d'une partition

Une partition comprend deux phases lors de son exécution, la première phase est appelée "phase de préparation", la seconde est la "phase opérationnelle". La figure 3.3 représente ces deux phases pour trois partitions P0, P1 et P2.

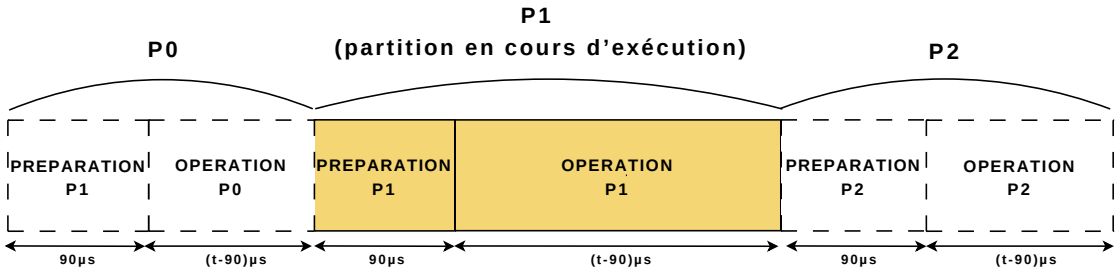

FIG. 3.3 – Les deux phases de trois partitions.

La phase de préparation dure 90 microsecondes pour chaque partition et est utilisée par le noyau pour exécuter certaines opérations afin que la prochaine partition fonctionne correctement : 1) sauvegarde du contexte de la partition précédente (celle qui a été exécutée avant P1, il s'agit de P0) ; 2) configuration de la MMU en ajoutant de

nouvelles entrées dans la TLB pour la prochaine partition (P1) et en supprimant les anciennes ; 3) configuration du prochain *Global Timer* à la durée de la phase opérationnelle (90 - durée totale d'exécution de la partition) microsecondes (ici, il s'agit de la phase opérationnelle de P1). Cette phase se termine par une mise en attente[31]. Cette attente permet de s'assurer que la durée d'exécution de la phase de préparation est bien 90μs.

Lorsque cette phase est terminée, la seconde phase (la phase opérationnelle) débute par la configuration du *Global Timer* pour la prochaine tranche de temps (ici, celle de la phase de préparation de P2) puis la restauration du contexte de la nouvelle partition courante (ici, P1). Enfin, la première instruction exécutée de la partition est à l'adresse du registre PC (*Program Counter*) qui vient d'être restauré. Pour mieux comprendre comment l'interruption est émise lors des changements de partition, la suite de cette sous-section traite des interruptions GT (*Global Timer*).

Lorsqu'une interruption de type *Global Timer* survient, une fonction du noyau est exécutée pour traiter l'interruption.

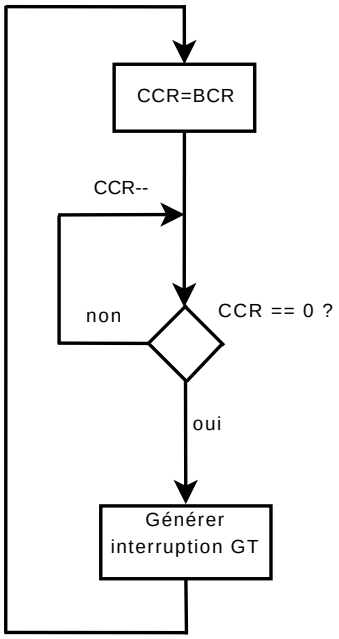

FIG. 3.4 – Fonctionnement d'un *Global Timer*.

Sur le schéma 3.4, deux registres sont utilisés pour les interruptions de type *Global Timer* : CCR pour le *Current Counter Register* et BCR pour le *Base Counter Register*. Le premier contient la valeur actuelle du compteur et le second contient la valeur maximale depuis laquelle le compteur sera décrémenté. Un troisième registre est utilisé, le VPR (*Vector/ Priority Register*) qui permet de définir la fonction qui sera invoquée lorsque

[31] Cette mise en attente est, en fait, une boucle infinie en mode utilisateur.

l'interruption surviendra. L'interruption est donc générée une fois que le compteur atteint la valeur zéro, ce qui conduit à exécuter une fonction du noyau (`nk_int_vector()`). Cette fonction invoque ensuite une autre fonction pour aller configurer le *Global Timer* pour les partitions suivantes.

3.2 Architecture du P4080

Le P4080 est un calculateur conçu pour être intégré dans des équipements de télécommunications. Il permet d'accélérer le traitement et le filtrage des paquets. Le P4080 respecte l'architecture QorIQ de Freescale. Il contient un processeur PowerPC de type RISC avec 8 cœurs e500mc pouvant atteindre une fréquence de 1.5GHz indépendamment les uns des autres, avec 3 niveaux de caches. Ils peuvent être exécutés de façon symétrique ou asymétrique. Le processeur peut fonctionner dans trois modes de privilèges différents : utilisateur, invité privilégié et hyperviseur. Le mode "utilisateur" (*user*) est le mode le plus restreint. Il est utilisé pour les applications qui ne doivent pas avoir d'accès privilégiés. Le mode "invité privilégié" (*supervisor*) est utilisé lorsqu'un moniteur de machines virtuelles est présent. Le mode "hyperviseur" (*hypervisor*) est le niveau le plus privilégié des trois et permet de gérer l'ensemble du matériel du calculateur grâce à des accès et des instructions non-restreints. La figure 3.5 représente l'architecture du P4080.

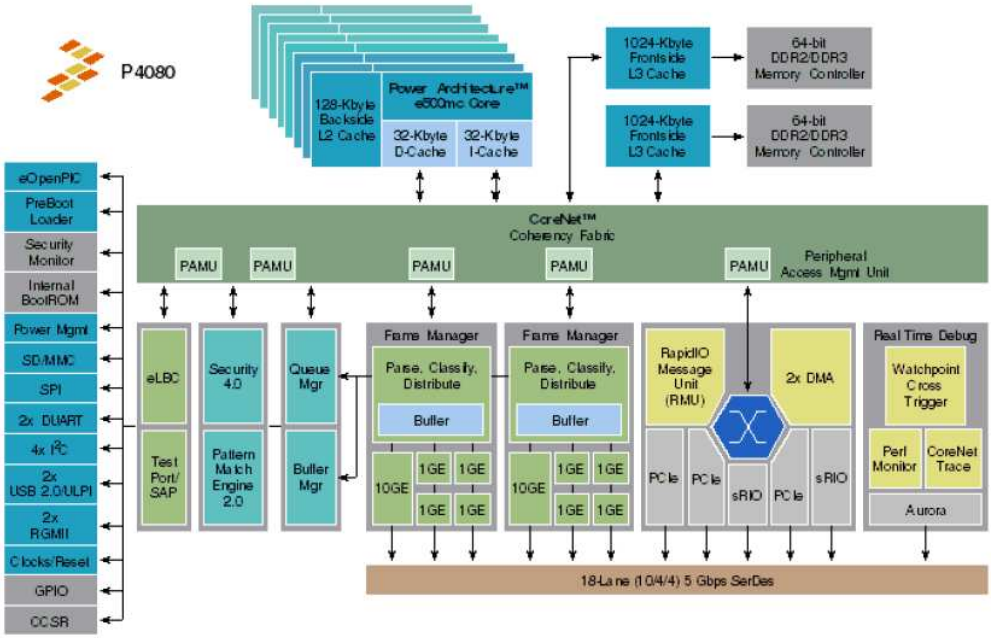

FIG. 3.5 – Architecture du P4080 (extrait de [Fre11]).

Sur le haut de la figure 3.5 sont représentés les 8 cœurs avec les 3 niveaux de caches. Ceux-ci sont connectés au bus CoreNet qui permet de relier simplement les périphériques avec la mémoire, le processeur et les différentes autres fonctions du P4080. Sur la gauche de la figure sont représentés les périphériques d'entrées/sorties : *Security Monitor*, SD/MMC, SPI, DUART, I2C, USB, etc. Le P4080 intègre également des PAMUs (*Peripheral Access Management Unit*) assimilables à des I/OMMUs (pour architecture Intel) pour se protéger contre des accès malveillants depuis un périphérique. La plateforme offre la possibilité d'utiliser un démarrage sécurisé (*secure boot*). Il est possible de signer numériquement le code qui est exécuté par le processeur pour s'assurer que celui-ci n'a pas été altéré. Un autre composant important est le DPAA (*DataPath Acceleration Architecture*), qui permet d'accélérer le traitement des paquets Ethernet. D'autres contrôleurs tels que le PCI Express ou le RapidIO sont également utilisables.

3.2.1 Structure des cœurs e500mc

Un cœur e500mc exécute des instructions 32 bits et implémente 32 registres généraux ainsi que 32 registres en virgules flottantes 64 bits. Le jeu d'instructions du e500mc est Power ISA version 2.06[32] (*Performance Optimization With Enhanced RISC Instruction Set Architecture*). Le cœur contient un cache L1 de 32 Ko pour les données et 32 Ko pour les instructions. Le cache L2 a une capacité de 128 Ko, il peut contenir à la fois des données et des instructions. Les deux caches sont protégés avec des codes correcteurs d'erreurs. Le e500mc contient une MMU dont l'objectif est de définir des droits d'accès à certaines portions de mémoire.

Le e500mc permet également de surveiller et compter des événements tels que les périodes des horloges de processeur, les omissions du cache d'instructions ou du cache de données ou encore les branches mal prédites (*mispredicted branches*). Les registres de surveillance de performance (*Performance Monitor Registers*) sont utilisés pour configurer et traquer les opérations de surveillance de performance. La gestion de l'alimentation permet de réduire la consommation d'énergie grâce à trois modes différents d'arrêts pour le cœur, sommeil léger, sommeil profond ou arrêt.

3.2.2 Contrôleur d'interruption

Le MPIC (*Multicore Programmable Interrupt Controller*) est l'unité permettant de contrôler les interruptions sur des architectures multi-cœurs. Elle est également responsable de recevoir les interruptions générées par le matériel depuis différentes sources internes et externes, les classer par priorité et livrer les signaux à la destination appropriée.

Le MPIC peut recevoir des interruptions de plusieurs sources :
- les signaux externes IRQ ;
- les sources d'interruptions internes (jusqu'à 112) ;
- les messages partagés entre programmes ;

[32] `https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf`

• interruptions générées dans le MPIC (*Global Timer*, *InterProcessor Interrupt*).

Les interruptions peuvent être routées vers n'importe lequel des 8 cœurs. Il est également possible de faire du *multicast* pour une diffusion simultanée sur plusieurs cœurs.

Le MPIC gère également des compteurs : les Global Timers. Ces compteurs permettent de générer une interruption lorsqu'un compteur décrémental (horloge) atteint la valeur zéro. Il suffit pour cela de lui donner une valeur correspondant à la durée souhaitée à attendre (registre GTBCRB0) puis de préciser quelle fonction appeler (registre GTVPRB0) lorsque la valeur atteint zéro.

3.2.3 *DataPath Acceleration Architecture* (DPAA)

Le DPAA permet de réduire les surcharges logicielles et d'accélérer le traitement des paquets Ethernet. Il est composé de cinq éléments : le *Security and Encryption Engine* (SEC), *Queue Manager* (QMan), *Pattern-Matching Engine* (PME), *Buffer Manager* (BMan) et le *Frame Manager* (FMan).
Ces éléments sont représentés dans la figure 3.6.

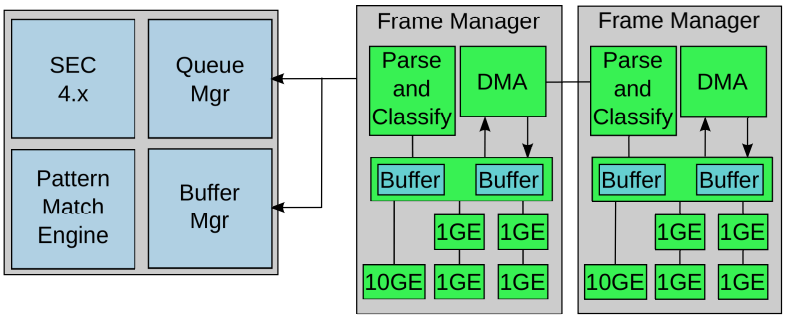

FIG. 3.6 – QorIQ DataPath Acceleration Architecture.

Le QMan est le composant qui permet le partage simplifié des interfaces réseau et des accélérateurs matériel par les cœurs. Il fournit aussi les moyens de partager des données entre cœurs pour être capable de paralléliser certaines tâches.

La première fonction du BMan est de réduire la surcharge de traitement par le logiciel grâce à l'utilisation de groupes de tampons (*pools of buffers*). Ces groupes de tampons sont mis à disposition des autres composants matériels et sont alloués et libérés par le BMan.

Le FMan permet de réaliser des analyses et classification configurables de trames entrantes avec pour but de sélectionner la file de trames d'entrée appropriée pour expédier le traitement vers un CPU ou un groupe de CPU.

Le SEC est un moteur d'accélération cryptographique. Il implémente les algorithmes de chiffrement par bloc, par flux, de hachage, asymétriques, générateur de nombres aléatoires et vérificateur d'intégrité d'exécution.

Le PME est une fonctionnalité de correspondance de modèles (*pattern-matching*) haute-performance permettant de rechercher des modèles dans des paquets réseau.

3.2.4 Démarrage sécurisé

Le P4080 intègre plusieurs mécanismes de sécurité : le bit de non-exécution (X bit de la MMU), l'hyperviseur, les PAMUs, le *Security Engine*, le contrôleur de débogage sécurisé, la détection d'un appareil de sécurité défaillant, le *Pre Boot Loader* (PBL), le fusible de sécurité (FSP), le moniteur de sécurité.

Le démarrage sécurisé utilise une clé publique RSA pour déchiffrer une empreinte SHA-256 signée et la comparer à une empreinte calculée sur la même portion de code. Ce code est en fait une image qui contient des instructions exécutables, des informations de configuration et une entête du fichier de séquence de commande. Il peut être en clair ou chiffré dans la mémoire. C'est le développeur qui décide du chiffrement ou non et du lieu où la clé privée RSA est stockée en mémoire. Elle ne doit en aucun cas être disponible à un tiers pour éviter la possibilité de générer une image alternative, ce qui rendrait inutile le démarrage sécurisé.

3.3 Présentation de la plateforme

Notre plateforme expérimentale comprend des éléments logiciels (le noyau temps-réel et l'IDE Codewarrior) et matériels (le P4080 et la sonde JTAG). Le calculateur QorIQ P4080 de Freescale a été choisi par Airbus car son processeur de type PowerPC et ses performances élevées permettent d'assurer une forte compatibilité avec les autres logiciels développés dans l'entreprise. Il permet également de proposer de nouvelles technologies (démarrage sécurisé, utilisation de plusieurs cœurs, etc.). Le noyau temps-réel est configuré statiquement (allocation de la mémoire pendant la compilation) ce qui permet de satisfaire la contrainte du déterminisme, et il n'y a aucune allocation de la mémoire pendant l'exécution du noyau. Cette configuration est réalisée via des fichiers dédiés. Un outil de génération automatique de code est utilisé pour générer le code du noyau et des interfaces entre le noyau et les applications à partir de la configuration.

Cette section est consacrée à la présentation détaillée de cette plateforme, en particulier, des mécanismes que nous avons utilisés pour pouvoir observer le comportement du noyau et des applications ainsi que la configuration de ces mécanismes. Cette observation est principalement basée sur l'utilisation d'une sonde JTAG et d'un logiciel approprié. La figure 3.7 propose une vue globale de notre plateforme d'expérimentation.

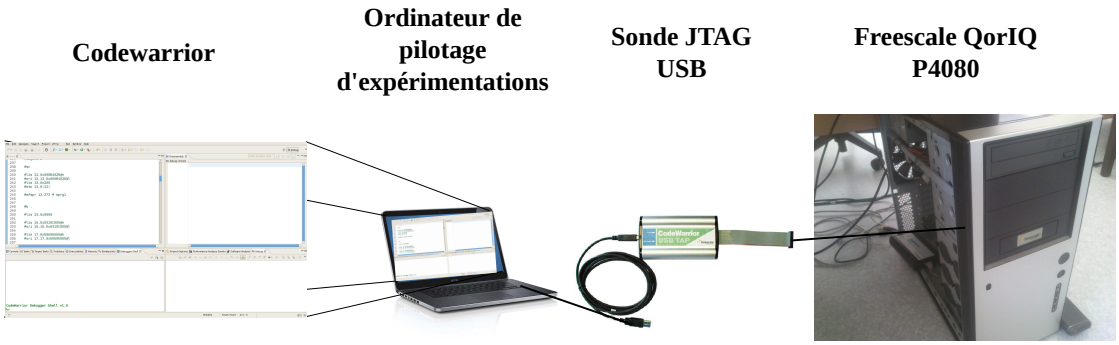

FIG. 3.7 – La plateforme d'expérimentation.

3.3.1 Observation du noyau et des applications

Pour notre étude, nous avons utilisé une version datant de 2010 du noyau expérimental d'Airbus. Cette version est actuellement en cours de développement mais elle permet néanmoins d'utiliser des fonctions du P4080 (notamment le MPIC et la MMU) avec le noyau et les partitions (séparation de la mémoire et décomposition en slots de temps). Pour être capable de tester efficacement le noyau et les applications, nous avons utilisé le logiciel Codewarrior (version 10.1.1). Il permet en effet de charger des applications sur une cible, de configurer du matériel, de contrôler et d'observer cette cible grâce à une sonde JTAG.

La sonde JTAG (voir figure 3.8) est l'interface entre le P4080 et le PC exécutant Codewarrior, elle permet de convertir les signaux de la carte en information compréhensible par le logiciel. Elle peut être connectée par le développeur à un ordinateur sur un port USB 2.0 ou un port Gigabit (disponibles par Freescale). Le second est bien sûr beaucoup plus rapide (de 100Mbits/s à 1Gbits/s) que le premier (quelques dizaines de Mbits/s) mais requiert une connexion au réseau local ce qui peut être une contrainte car cela signifie que la sonde est directement accessible à n'importe quel personne (honnête ou non) qui connaît l'adresse IP de la sonde[33]. Pour nos expérimentations, nous avons utilisé la sonde JTAG avec un port USB.

Le port JTAG se situe sur la carte mère du P4080. La sonde JTAG est connectée à ce port et à un ordinateur disposant de Codewarrior à l'autre extrémité. Comme pour tout programme de débogage, l'exécution peut se faire en mode pas-à-pas ou en mode continu jusqu'à ce que le programme rencontre un point d'arrêt.

Cette sonde permet par exemple de débugger le code en cache, ROM, RAM et dans la mémoire flash, d'afficher et de modifier la mémoire sur la cible ou encore d'examiner

[33]Cette méthode est donc plutôt utilisée pour un ensemble de développeur ayant accès à la même plateforme.

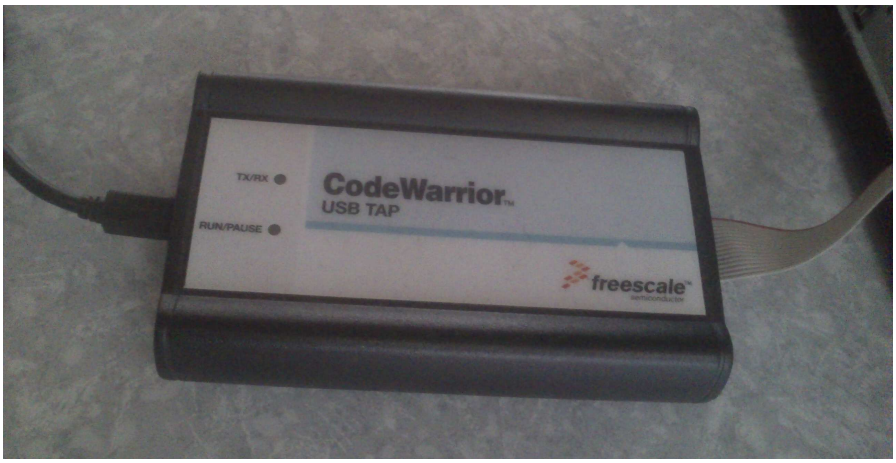

FIG. 3.8 – Sonde JTAG de Freescale.

et de modifier n'importe quel registre du processeur. Il supporte également toutes les vitesses des cœurs du processeur. L'avantage avec le port USB est qu'il ne nécessite aucune configuration de la sonde contrairement à l'interface réseau Gigabit qui nécessite forcément l'attribution d'une adresse IP pour être accessible sur le réseau. Codewarrior détecte automatiquement le branchement de la sonde USB. Il est aussi possible de brancher plusieurs sondes USB sur le même ordinateur et d'en spécifier une seule à utiliser dans Codewarrior (grâce à son numéro de série).

3.3.2 Configuration de la plateforme d'expérimentation

La configuration de la plateforme d'expérimentation est basée sur le paramétrage du P4080 lui-même, sur l'installation et la paramétrage des composants logiciels du système avionique (le noyau et les applications), ainsi que sur le paramétrage des outils de contrôle et d'observation (la sonde JTAG et l'application Codewarrior). Nous présentons ces différentes configurations dans les sous-sections suivantes.

3.3.2.1 Configuration de Codewarrior et de la sonde JTAG USB

L'interface graphique de Codewarrior offre plusieurs vues : le code source (C ou assembleur), la mémoire désassemblée, les *breakpoints* et les commandes de *debug* (pour continuer l'exécution jusqu'au prochain point d'arrêt ou faire une pause). La figure suivante 3.9 est une capture d'écran de Codewarrior version 10.1.1.

De nombreuses fonctionnalités sont offertes par ce logiciel. La plus intéressante est probablement le *debugger shell* qui nous permet de lire des registres et de la mémoire, de gérer les points d'arrêts et de créer des scripts.

Codewarrior offre la possibilité de configurer un ou plusieurs cœurs du processeur.

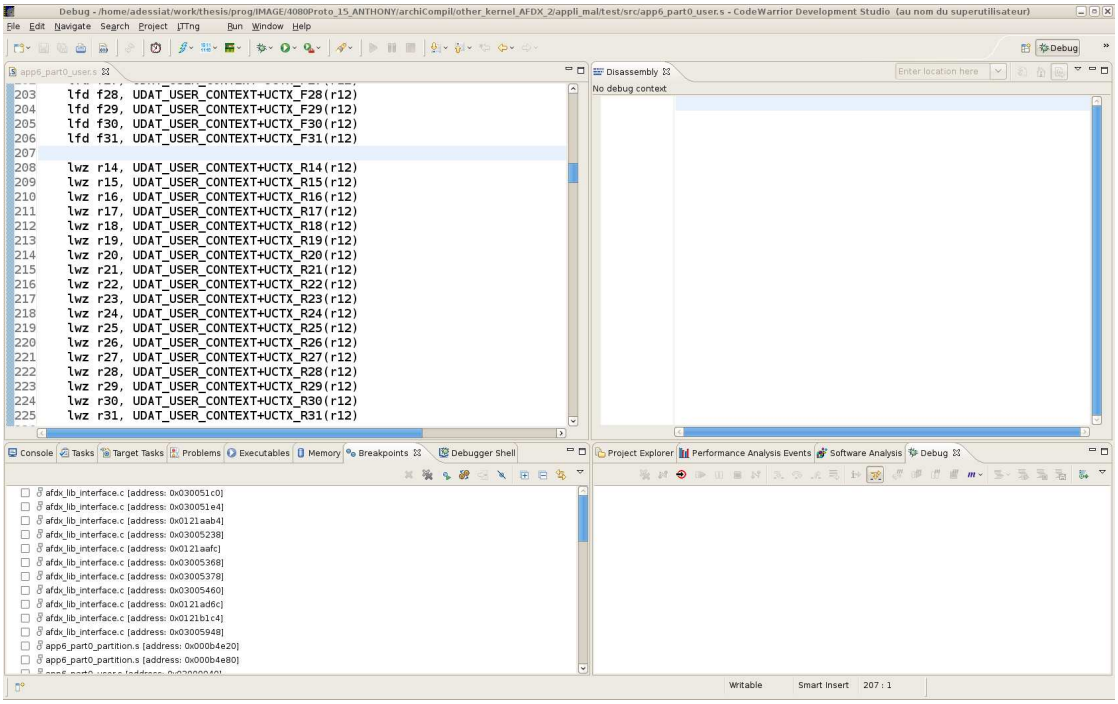

FIG. 3.9 – Capture d'écran du logiciel IDE Codewarrior de Freescale.

Nous n'utilisons qu'un seul cœur pour exécuter notre code pour nous placer dans les mêmes conditions que celles d'Airbus. Ensuite, la configuration du cœur lui-même s'effectue au démarrage du noyau. Cette initialisation est réalisée via un fichier de configuration. Ce fichier contient la configuration de plusieurs éléments importants du P4080 : les fenêtres d'accès local (*Local Access Window* ou LAW), la DDR et ajoute une entrée dans la MMU. Un ensemble de 32 LAWs[34] peut être configuré. Un LAW permet d'assigner une zone d'adresses à un périphérique. Pour notre plateforme, cinq de ces LAWs sont configurés pour Codewarrior : pour l'utilisation de la DDR1 et DDR2, PCI Express 3, et deux eLBC (ex : SRAM, ZBT RAM ou EEPROM). La taille de ces zones, l'adresse de base, la cible de la transaction sont précisées dans ce fichier de configuration.

La sonde JTAG USB ne nécessite aucune configuration particulière. Il suffit de la brancher à l'ordinateur pour qu'elle soit directement reconnue.

3.3.2.2 Configuration du P4080

Avant d'utiliser la plateforme, une configuration logicielle et matérielle s'impose. La configuration matérielle est réalisée grâce à des cavaliers (les SW[35]) présents sur la carte mère du P4080. Ils sont au nombre de 80. Nous pouvons modifier les paramètres sui-

[34] Une zone peut être définie sur 4Ko au minimum ou 64Go au maximum.
[35] SWitches.

vants : la vitesse de l'horloge interne (de 66.666 MHz à 133.333 MHz), le voltage des cœurs et de la plateforme ou encore les protections en écriture des configurations RCW.

La configuration logicielle du P4080 s'effectue en deux temps. Tout d'abord, un fichier (le PBL[36]) est chargé depuis la mémoire flash pendant le démarrage notamment pour configurer la fréquence du processeur. Ensuite, le noyau configure plusieurs éléments :

- les permissions de la zone de mémoire (MMU) ;
- le compteur de temps (*Time Base*) ;
- les interruptions, qui sont initialisées (MPIC) ;
- les caches, qui sont initialisés.

Dans les premières instructions de l'exécution du code noyau, quatre nouvelles zones de mémoire sont ajoutées dans la MMU : le CCSR[37] sur 16 Mo (nous y reviendrons en détail plus tard), le code noyau sur 1 Mo, les données du noyau sur 1 Mo puis les données d'instrumentation du noyau sur 1 Mo. Le compteur de temps est aussi activé (car il est désactivé par défaut) dans les premières instructions pour être capable de mesurer les durées des exécutions ce qui nous est nécessaire pour nos expérimentations. Les interruptions sont également utilisées pour permettre de gérer efficacement les événements et les erreurs sur la plateforme. Les IVORs (*Interrupt Vector Offset Registers*) permettent par exemple de définir les actions à entreprendre (adresse des instructions à exécuter) lorsqu'un événement survient (erreur de données, erreur d'alignement, erreur de données TLB, appel-système, etc.). Les caches sont utilisés pour accélérer la vitesse de transmission des données et des instructions entre le processeur et la mémoire principale.

3.3.2.3 Configuration du noyau

La configuration du noyau est réalisée par l'intermédiaire de plusieurs fichiers textes. Ces fichiers sont utilisés lors de la génération du noyau. Dans ces fichiers de configuration, nous retrouvons notamment la durée d'un cycle d'exécution, le nombre d'applications qui sont exécutées par le noyau, la durée maximale d'un appel d'une fonction distante (IPC) ou encore les actions à entreprendre (arrêt, redémarrage ou arrêt après trois redémarrages) lorsqu'une erreur survient.

3.3.2.4 Configuration des applications

La configuration des applications se fait également par des fichiers modifiables avant la compilation. Une application est composée d'une ou plusieurs partitions (voir le chapitre 1 sur les systèmes avioniques pour plus de détails). Chaque partition a donc une tranche de temps qui lui est réservée ainsi qu'un espace mémoire limité. Ces paramètres sont différents de ceux du noyau car celui-ci n'a pas de tranches de temps allouées. Il y a nécessairement la durée d'exécution de la partition ainsi que les zones de mémoire accessibles depuis la partition. Il est aussi nécessaire de préciser à quel moment (décalage en microsecondes) la partition doit commencer.

[36] *Pre-Boot Loader*
[37] *Configuration, Control and Status Registers.*

Après la présentation des différents composants de notre plateforme d'expérimentation, la section suivante est consacrée à la présentation détaillée de plusieurs attaques que nous avons pu réaliser sur notre plate-forme, cela concerne les attaques :

- ciblant la gestion de la mémoire,
- ciblant les communications,
- ciblant la gestion du temps,
- ciblant les mécanismes de tolérance aux fautes.

3.4 Attaques ciblant la gestion de la mémoire

Une corruption de la gestion de la mémoire peut permettre à une partition malveillante d'accéder à des données sensibles utilisées par le noyau ou par d'autres partitions. Nous décrivons dans l'exemple suivant comment provoquer une attaque par déni-de-service en abusant la gestion de la mémoire. Comme nous l'avons déjà indiqué, la plateforme P4080 est configurée en partie grâce à la zone *Configuration, Control, and Status Registers* (CCSR). Cette zone de 16 Mo contient la configuration de tous les composants matériels (MPIC, périphériques, contrôleurs réseau, alimentation, etc.). Or, nous avons découvert, en analysant le code source, que dans la version actuelle du système en notre possession, les accès en lecture et écriture à cette zone sont positionnés pour le noyau mais également pour les partitions. Cette configuration a été choisie délibérément pour que les différentes partitions puissent utiliser le contrôleur réseau AFDX sans utiliser de pilote partagé.

Cette configuration est particulièrement risquée car cela implique que les partitions peuvent modifier la configuration de tous les composants matériels de la plateforme. Par exemple, avec cette configuration, une partition malveillante est parfaitement capable de remplir les registres de la zone CCSR avec des données aléatoires. Pour nos expérimentations, nous avons choisi d'injecter, depuis la partition malveillante, des valeurs particulières dans cette zone afin de provoquer un déni-de-service. A titre d'illustration, un exemple de déni-de-service est présenté dans la prochaine sous-section, en utilisant des registres du CCSR et visant plus spécialement le Security Engine (SEC du DPAA) du P4080.

3.4.1 Attaques ciblant le *Security Engine*

Comme nous l'avons déjà vu précédemment, le SEC permet d'utiliser des mécanismes de chiffrement tels que AES, DES, RSA, etc., au sein du P4080, que ce soit pour le traitement des paquets Ethernet ou pour un autre usage. Il est géré par le *Security Monitor* qui est en charge de vérifier les erreurs du système et contrôler les clés cryptographiques utilisées pour le démarrage sécurisé. Par défaut, le gestionnaire de sécurité (*security monitor*) est dans un état *non-secure*, c'est-à-dire que les clés cryptographiques ne sont pas initialisées. La figure 3.10 montre les différents états possibles du gestionnaire de sécurité et les relations entre eux.

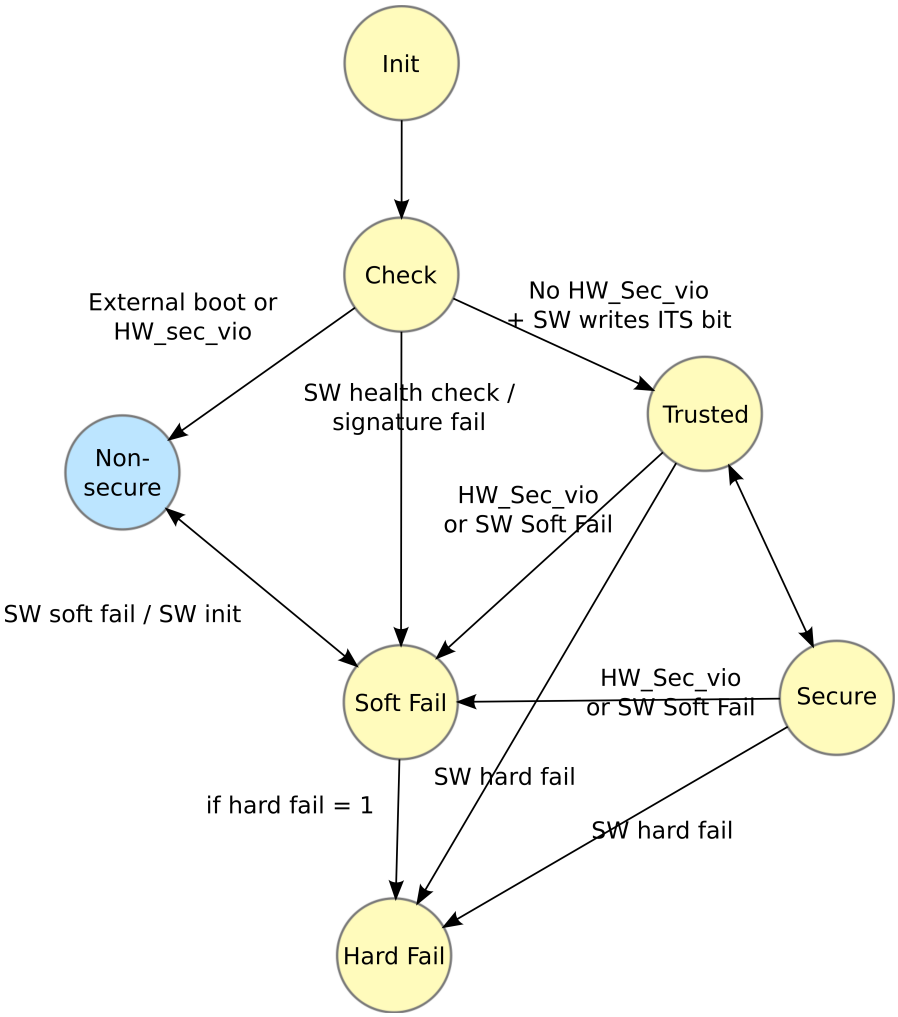

FIG. 3.10 – Diagramme des états du SEC [Fre11].

Le choix du démarrage sécurisé ou non se fait grâce au bit ITS (*Intent To Secure*) du registre `SFP_OSPR` (*OEM Security Policy Register*) des registres *Security Fuse Processor* (SFP). Pendant le démarrage du système, l'état du gestionnaire de sécurité passe successivement de *init* à *check* puis à *non-secure* si le bit ITS n'est pas positionné, ou *secure* si ce bit est positionné. Si une erreur logicielle est détectée lors du démarrage du système, le gestionnaire de sécurité entre dans un état "failed" ce qui bloque l'utilisation des clés cryptographiques et les positionne à zéro. Ce type d'erreur se manifeste par le positionnement à 1 du bit `SW_FSV` du registre `SM_HP` (adresse 0xFE314004). Si, à partir de cet état, une erreur matérielle est détectée, le système redémarre. Cette erreur se ma-

nifeste par le positionnement à 1 du bit `HAC_EN` du même registre `SM_HP` (tel qu'indiqué dans la spécification du P4080 [Fre11]). Ce registre est positionné dans le CCSR, zone mémoire dont les accès en lecture et écriture sont autorisés pour toutes les partitions, comme nous l'avons vu précédemment. En conséquence, si une partition malveillante modifie intentionnellement ces deux bits et les positionne à 1, la plateforme redémarre. Nous avons ainsi provoqué ce déni de service en modifiant ces 2 bits.

Un exemple similaire peut être réalisé en utilisant le *Run Control/Power Management* (RCPM). Nous le présentons dans la sous-section suivante.

3.4.2 Attaques ciblant le *Run Control/Power Management*

Le RCPM (*Run Control/Power Management*) est une autre zone du CCSR. Le RCPM gère l'alimentation des cœurs et des périphériques. Deux modes de faible alimentation sont possibles pour un cœur. Il s'agit des modes : assoupi (*dozing*) et endormi (*napping*). Le mode assoupi permet de suspendre l'exécution d'instructions, sans arrêter le fonctionnement des caches L1 et L2. Dans le mode endormi, toutes les horloges internes sont arrêtées et les caches L1/L2 ne sont plus fonctionnels. Deux registres sont utilisés pour passer dans ces modes. Le registre `RCPM_CDOZCR` (*Core Doze Control Register*) permet de passer en mode *core dozing* et le registre `RCPM_CNAPCR` (*Core Nap Control Register*) permet de passer en mode *core napping*. Dans la mesure où l'accès en écriture est autorisé pour toutes les partitions dans le CCSR (et donc dans la zone RCPM), il est donc possible pour une partition malveillante de modifier ces registres et donc de configurer les modes d'alimentation des cœurs, et ainsi de provoquer un déni-de-service en les positionnant en mode assoupi ou endormi.

3.5 Attaques ciblant les communications

Le noyau expérimental fourni par Airbus autorise plusieurs types de communications : une communication entre modules via le réseau AFDX (*Avionics Full DupleX*) et une communication entre partitions d'un même module via les IPC.

3.5.1 Attaques ciblant les communications AFDX

Pour communiquer entre les modules et échanger des informations, les systèmes avioniques utilisent principalement le réseau AFDX. Celui-ci est déterministe et basé sur Ethernet. Ses principaux avantages sont sa rapidité et sa simplicité (pour l'accès au réseau et l'évolution de l'architecture). Pour cela, le réseau utilise des liens virtuels (*Virtual Links*). Un *Virtual Link* (VL) caractérise un flot de données entre un émetteur et un ou plusieurs destinataires ; ce flot de données doit respecter une bande passante définie par configuration. L'identifiant du VL est contenu dans l'adresse MAC destination. D'autres informations comme le numéro de partition, le numéro de l'hôte et du réseau sont eux contenus dans l'adresse IP source, qui, même si elle est codée sur la même taille qu'une adresse IP usuelle, contient des informations légèrement différentes (le numéro de partition). Deux types de ports AFDX sont utilisés :

- *Service Access Points* (SAP) : les ports SAP ont été créés pour répondre à un besoin Ethernet “classique” pour les applications du monde ouvert (TFTP notamment).
- AFDX communication : chaque port implémente un service d'envoi de données et deux services de réception (en mode *sampling* et *queuing*)
  - mode *Sampling* : seule la dernière valeur est présentée à l'application par le driver.
  - mode *Queuing* : toutes les valeurs sont présentées à l'application par le driver.

La configuration de AFDX se fait avant la compilation des partitions car elle est statique et tous les paramètres doivent être définis avant exécution. Nous allons détailler comment réaliser cette configuration AFDX.

3.5.1.1 Configuration des tables AFDX

La configuration des tables AFDX se fait grâce à deux fichiers contenant deux tableaux avec tous les paramètres nécessaires : VL, BAG[38], *Max frame size*, redondance, numéro de réseau, etc. Un outil codé en Perl appelé `Matrix` (produit et mis à notre disposition par Airbus) utilise ces fichiers pour générer du code C qui est ensuite compilé pour obtenir le fichier ELF exécutable correspondant au pilote AFDX pour la partition. La figure 3.11 représente la chaîne de compilation pour une partition utilisant l'AFDX. Il est nécessaire d'utiliser l'outil Matrix pour générer des fichiers C, ensuite, nous utilisons le compilateur GCC pour les compiler avec le programme principal.

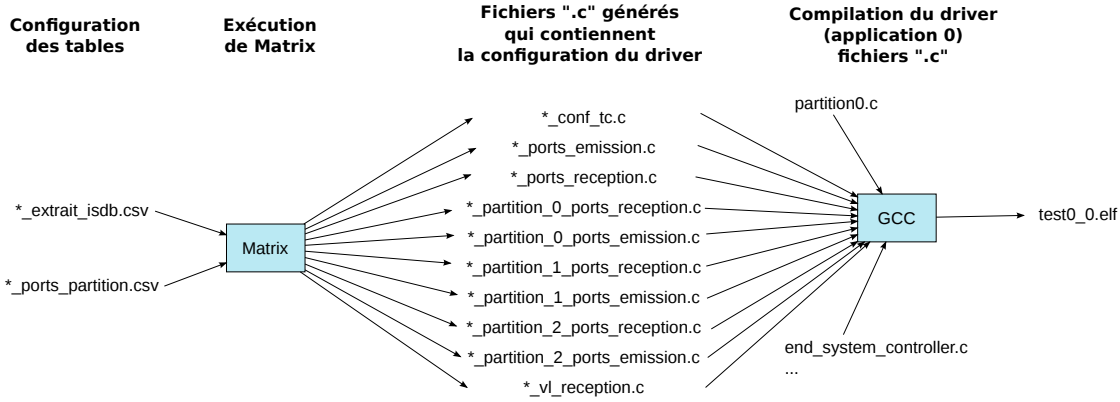

Fig. 3.11 – Chaîne de compilation pour une partition utilisant AFDX.

Pour réaliser nos tests AFDX, nous avons utilisé une carte AFDX d'Airbus, connectée en PCI avec un pont PCI vers PCI Express. Cette carte est par défaut “ouverte” car elle est destinée à réaliser des tests. C'est la raison pour laquelle nous avons détecté certains problèmes évidents de sécurité comme la gestion des accès DMA par exemple. Cependant nous avons également identifié d'autres vulnérabilités qui ne sont pas uniquement liées à

[38] Le *Bandwidth Allocation Gap* (BAG) représente le délai minimum entre 2 trames consécutives d'un VL (de 1ms à 128ms par valeurs multiples de 2).

cette configuration en mode test. Nous présentons l'ensemble de ces vulnérabilités dans la suite.

3.5.1.2 Vulnérabilités détectées

La première vulnérabilité découverte est liée à la configuration des accès DMA, puisqu'ils sont autorisés en lecture et écriture de façon illimitée à toute la mémoire (l'envoi et la réception de données nécessite en effet de réaliser des accès DMA, comme il est coutume avec une carte Ethernet classique). Cette configuration est réalisée dans le CCSR, et comme nous l'avons vu précédemment, la gestion des droits d'accès au CCSR fait que toute partition peut modifier tous les registres présents dans cette zone. Une partition malveillante peut donc tout à fait modifier la gestion des accès DMA en accédant en écriture au CCSR. Il lui est donc tout à fait possible d'envoyer des commandes à la carte AFDX sans aucun contrôle préalable, et de faire n'importe quelle lecture ou écriture sur toute la mémoire du système.

Une seconde vulnérabilité concerne l'accès à certaines variables globales utilisées pour les communications AFDX. Ces variables sont créées dans une zone de mémoire partagée entre les applications qui effectuent des communications AFDX, et donc modifiables par n'importe laquelle de ces partitions. En conséquence, une partition malveillante qui utilise le réseau AFDX a donc accès à cette zone de mémoire et peut donc modifier ces variables. Nous avons ainsi pu montrer qu'il est possible de bloquer depuis une partition malveillante les envois AFDX d'autres partitions.

La troisième vulnérabilité concerne une variable appelée `AFDX_PARTITION_ID` qui permet d'identifier une partition lors de l'envoi et la réception de paquets. Cette variable est locale à chaque partition et est stockée dans son espace mémoire dédié. Mais on peut imaginer qu'une partition malveillante modifie cette variable dans le but de recevoir des données qui ne lui sont pas destinées ou de se faire passer pour une autre partition lorsqu'elle envoie des données. Si une partition malveillante modifie cette variable en lui affectant un numéro de partition invalide, cela provoque une erreur. Si elle utilise un numéro valide d'une autre partition, alors le paquet émis par la partition malveillante sera considéré émis par cette autre partition. Lors de la réception de paquet, il est également possible à une application malveillante de modifier cette variable pour tenter de recevoir des paquets destinées à une autre partition mais la propriété de déterminisme du réseau AFDX fait qu'en pratique, il est impossible que des données soient reçues à un instant t pour une partition i alors que c'est la partition j qui est en cours d'exécution. Nous considérons donc que cette vulnérabilité est beaucoup plus difficile à exploiter en réception. Dans le cadre de nos expérimentations, nous avons effectivement réussi à modifier cette variable au moment de l'envoi et donc à créer une partition qui émet des paquets en se faisant passer pour une autre partition.

La quatrième vulnérabilité concerne les partitions *Data Loader* et *Flash Manager* qui permettent de télécharger et de recopier des données (en particulier de mises à jour). Dans la version actuelle de ces partitions, la *Data Loader* ne fait aucune vérification sur les données envoyées au *Flash Manager*. Si par conséquent, ces données contiennent des informations erronées concernant des données de configuration des partitions ou

des coeurs du processeur par exemple, celles-ci seront mises à jour et pourront considérablement modifier le comportement global du système jusqu'à le rendre totalement inopérant. Cette dernière vulnérabilité a été découverte grâce à l'analyse de code source des applications mais elle n'a pas donné lieu à des expérimentations.

3.5.2 Attaques ciblant les IPC

Notre noyau temps-réel expérimental implémente les IPC (*Inter-Process Communications*) qui permettent à une partition (dite locale dans la terminologie IPC) d'appeler une fonction fournie par une autre partition (dite distante). Pour cela, (1) la partition locale exécute un appel-système pour que le noyau exécute une commutation de contexte ; (2) le noyau change le contexte de la partition locale pour le contexte de la partition distante ; (3) la fonction distante est exécutée pendant 27 microsecondes puis exécute un appel-système pour que le noyau exécute une commutation de contexte ; (4) le noyau recharge le contexte de la partition locale et (5) la partition locale continue à être exécutée. La figure 3.12 représente un exemple de communication IPC.

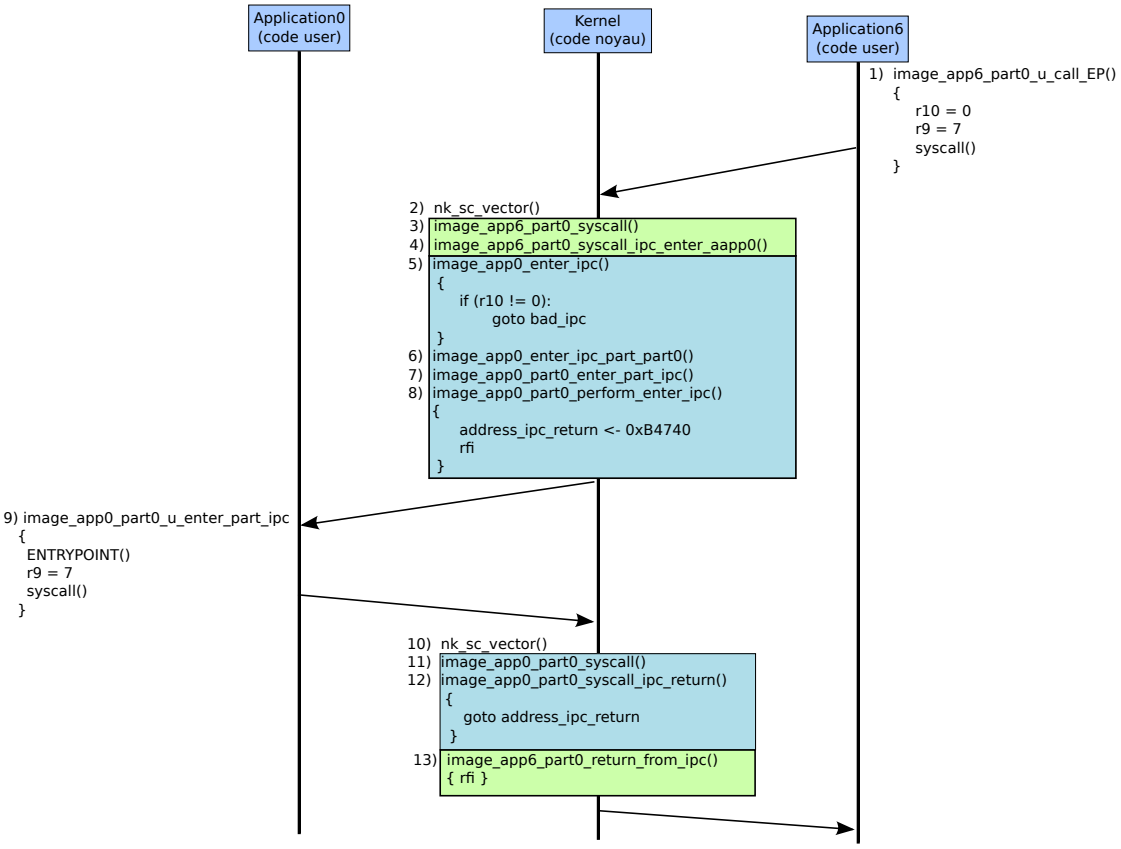

Fig. 3.12 – Exemple de communication IPC

Deux appels système sont créés pour la gestion des IPC : un appel-système est créé dans la partition locale pour appeler la fonction distante et un autre dans la partition distante pour revenir dans la partition locale. Ces appels système permettent de donner la main au noyau car c'est lui seul qui a les privilèges suffisants pour passer d'une partition à une autre (modification de contexte). Deux registres généraux (`GPR9` et `GPR10`) sont initialisés avant l'appel-système pour appeler la fonction distante. `GPR9` permet de préciser le numéro de l'appel-système et `GPR10` est utilisé par le noyau pour une action non-définie car toujours en cours de développement.

L'application 6 correspond à la partition locale et l'application 0 correspond à la partition distante. Nous allons détailler maintenant trois problèmes d'implémentations du noyau qui peuvent conduire à des vulnérabilités (en particulier des dénis-de-service) si la partition locale ou distante n'agit pas exactement comme elle est supposée le faire.

3.5.2.1 Modification de l'ordre des appels-système

Une première tentative d'attaque consiste pour la partition distance à exécuter l'appel-système lui permettant de revenir dans la partition locale alors que la partition locale n'a pas encore effectué d'IPC en invoquant elle-même son propre appel système. La figure 3.13 représente cet exemple.

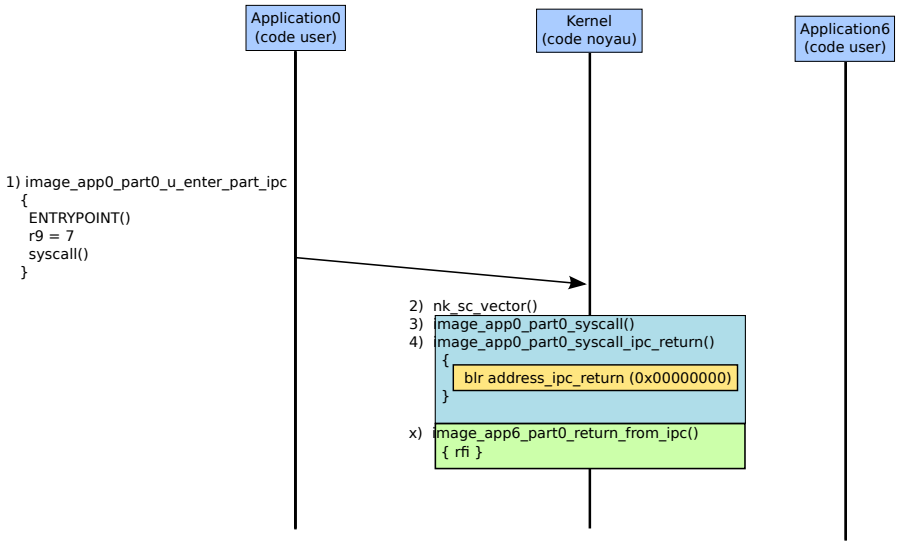

Fig. 3.13 – Appel-système depuis l'Application 0

Dans ce cas, le code noyau qui vérifie la source de l'appel-système va provoquer une erreur de type ITLB (*Instruction Translation Lookaside Buffer*) en exécutant l'instruction à l'adresse 0x00000000 qui se situe dans un espace mémoire privilégié (et qui provoque un déni-de-service). Là encore, c'est l'implémentation qui n'est pas complète et qui ne devrait pas pointer vers cette adresse.

3.5.2.2 Interruption dans la fonction distante

Dans cette seconde attaque, nous avons voulu observer le comportement du noyau lorsque la durée accordée à l'exécution de l'IPC (27 microsecondes) se termine avant que toute la fonction n'ait été exécutée. Pour cela, nous avons défini une boucle infinie dans la fonction distante pour qu'une interruption de type *Global Timer* puisse provoquer le retour dans le noyau au lieu de l'appel-système. Dans ce cas, l'exécution continue dans une boucle infinie (figure 3.14), l'implémentation actuelle ne l'a pas pris en compte.

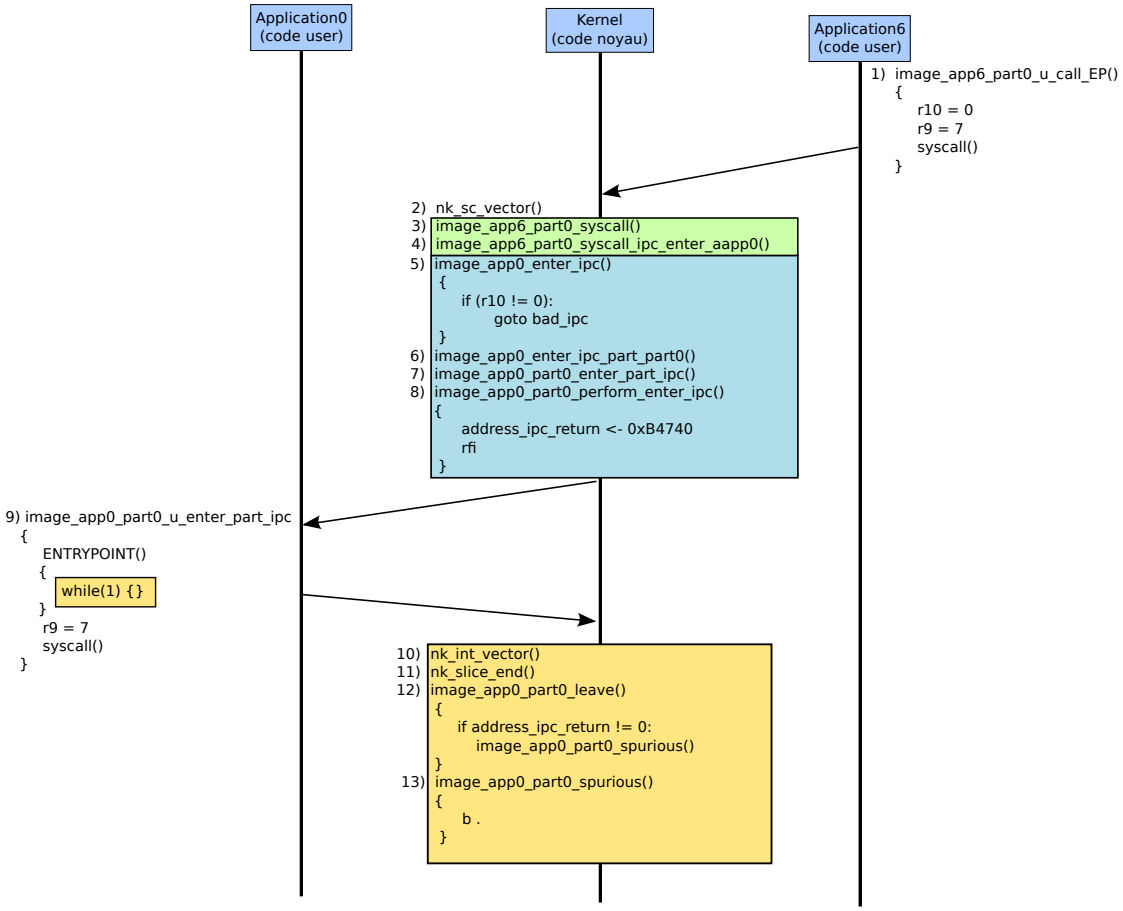

FIG. 3.14 – Boucle infinie l'Application 0

3.5.2.3 Modification du registre GPR10

Cette dernière attaque concerne la modification intentionnelle du registre GPR10 dans le but de rendre le système indisponible. En effet, ce registre `GPR10` est initialisé à la valeur 0 dans le code de la partition (et donc en mode utilisateur) avant de réaliser l'appel-système pour appeler la fonction distante. Il est ensuite comparé à 0 dans le code

noyau. Si une valeur différente est présente, alors une boucle infinie est exécutée sinon le code poursuit son exécution.

Si la partition locale est malveillante alors elle peut positionner ce registre `GPR10` à une valeur différente de 0 pour que le noyau rentre dans la boucle infinie. Dès qu'une boucle infinie dans le noyau est exécutée, il n'est plus possible de sortir de cette boucle par un autre moyen que le redémarrage ou une interruption `NMI` car le noyau ne peut pas être interrompu par une interruption "classique". La présence de cette boucle à cet endroit ne devrait pas être possible, il s'agissait en fait d'une implémentation incomplète car le noyau est en cours de développement.

Même si globalement ces résultats d'expérimentations sont principalement dues au fait que le noyau est en cours de développement, elles ont permis de mettre en évidence des comportements qui n'avaient pas été considérés comme possibles tant que l'on envisage pas des actes malveillants. Modifier volontairement le séquencement d'un IPC ou inclure une boucle infinie dans son propre code sont des comportements malveillants qu'il est donc important de prendre en compte lors du développement, sous réserve de provoquer des dénis-de-service du système.

3.6 Attaques ciblant la gestion du temps

Cette section est consacrée à la présentation des expérimentations que nous avons menées pour perturber la durée d'exécution d'une partition, tout d'abord en utilisant les interruptions, ensuite en modifiant la configuration d'une partition.

3.6.1 Première attaque : utilisation des interruptions

La durée d'exécution des tranches de temps pour chaque partition est basée sur le calcul du WCET (*Worst Case Execution Time*) qui est stocké dans les fichiers de configuration de chaque partition. Le MPIC fournit la gestion des interruptions multiprocesseurs et est responsable d'envoyer des interruptions générées par le matériel (internes et externes), leur associer des priorités, et les envoyer aux cœurs appropriés.

Le principe de l'attaque à laquelle nous avons pensé consiste à essayer de faire en sorte que l'exécution d'une partition malveillante déborde de sa tranche de temps allouée, de façon à réduire le temps d'exécution de la partition suivante. Si cette partition est une partition critique, cela peut donc avoir un impact sérieux. Cette attaque nécessite l'utilisation d'une interruption, ainsi que nous le présentons dans la sous-section suivante.

3.6.1.1 Réalisation du programme

Pour réaliser le programme de notre propre partition malveillante, nous avons créé une boucle et ajusté une valeur de sortie de boucle pour provoquer une interruption au plus proche de la fin de la durée d'exécution. Cette interruption est une instruction

illégale (instruction `rfi`[39] dans notre cas) qui provoque un retour dans le noyau pour être traité. Notre objectif est ainsi de pouvoir consommer du temps CPU dans le noyau (qui est non interruptible) grâce à cette interruption déclenchée au dernier moment dans notre partition malveillante, et ainsi voler du temps CPU à la partition suivante.

3.6.1.2 Instrumentation du noyau

Nous avons modifié le noyau pour être capable de sauvegarder en mémoire les durées d'exécution des partitions P6 et P7 (phase de préparation de P7 et phases opérationnelles de P6 et P7) : les temps T0, T1, T2 et T3. La figure 3.15 représente ces durées pour chaque partition (la phase de préparation est notée W et la phase opérationnelle est notée P suivi du numéro de la partition) :

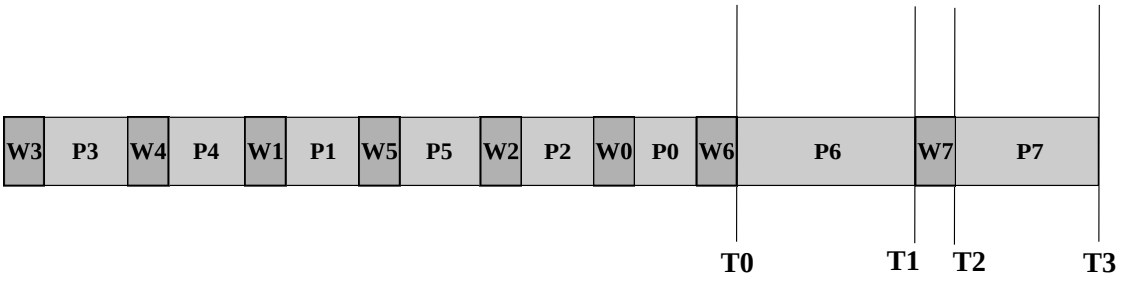

FIG. 3.15 – Mesures des temps sur un cycle (5 ms).

Nous sauvegardons ces 4 temps[40] pendant 1024 cycles puis nous analysons les résultats. A noter que ces temps correspondent aux réelles durées d'exécution des différentes phases et non au compteur du *Global Timer* qui déclenchent les interruptions une fois qu'il vaut zéro. Pour cela, nous avons réalisé un programme en Python qui, à partir de ce tableau de 1024 cycles, permet de calculer toutes les durées d'exécution de la partition P6 et P7 ainsi que celle de la phase de préparation de la partition P7.

3.6.1.3 Résultats obtenus

Nous prenons comme exemple deux partitions P6 (malicieuse) et P7 (non-malicieuse) avec leurs phases de préparation associées W6 et W7. Les durées des phases de préparation sont fixées à 90 microsecondes pour toutes les partitions et les durées des phases opérationnelles sont fixées à 1410 microsecondes (soit un total de 1500 microsecondes pour chacune des deux partitions). La figure 3.16 représente les durées d'exécutions des différentes phases de ces deux partitions ainsi que les compteurs CCR (*Current Count Register*) du MPIC dont la valeur décrémente jusqu'à la valeur 0.

[39] L'instruction *return from interrupt* (rfi) permet de retourner en mode utilisateur lorsque l'on est dans le noyau et doit donc être seulement utilisé dans le noyau.

[40] Utilisation des registres `TB` (Time Base) et `TBL` (Time Base Lower).

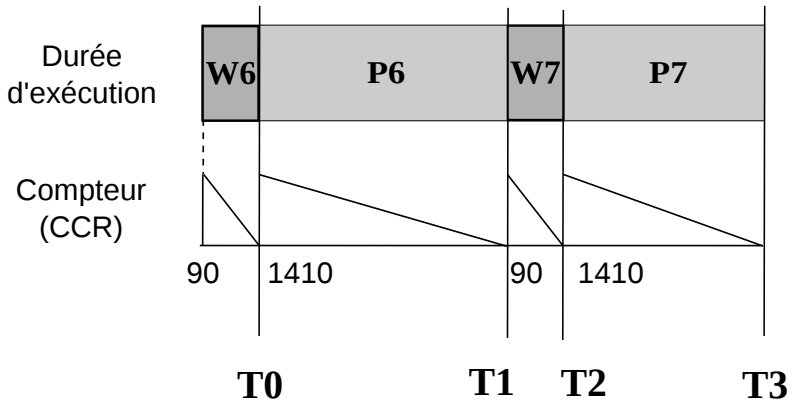

FIG. 3.16 – Comportement normal des deux partitions.

Une attaque sur la gestion du temps consiste pour une partition non-critique malveillante, à être capable de réduire la durée d'exécution d'une partition critique. Comme nous l'avons expliqué auparavant, le principe consiste donc, pour une partition malveillante, à augmenter sa tranche de temps grâce au déclenchement volontaire, à la fin de sa tranche de temps, d'interruptions qui sont gérées par le noyau (qui ne peut être interrompu) (voir figure 3.17).

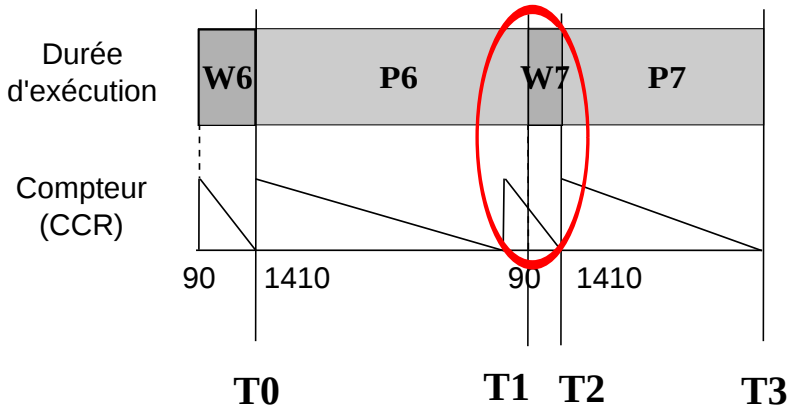

FIG. 3.17 – Dépassement de P6.

Lorsque la partition P6 enclenche une exception à la fin de sa tranche de temps, nous avons été capable d'augmenter sa durée de 6 microsecondes et donc de réduire la durée de la partition P7 de 6 microsecondes. Pour cette expérience, les conséquences n'ont pas été significatives car la marge de sécurité prise par le noyau est suffisante mais dans d'autres contextes, il est tout à fait possible d'imaginer que cette réduction puisse conduire à défaillance.

3.6.2 Seconde attaque : modification de la configuration d'une partition

Un autre type d'attaque a été envisagé, même s'il est beaucoup plus difficile à réaliser en pratique. Il consiste à essayer de faire en sorte que la partition P7 nécessite un temps de préparation supérieur à la durée de 90μs prévu statiquement. Pour cela, nous avons considéré une partition P7 qui utilise énormément de mémoire et donc d'entrées dans la MMU. Dans ce cas, la phase de préparation, durant laquelle notamment ces entrées MMU sont préparées, peut durer plus de 90μs. Nous avons ainsi ajouté volontairement plusieurs dizaines d'entrées dans la MMU de P7, ce qui allonge donc la durée de la phase de préparation, car il s'agit de code en mode noyau non interruptible.

A titre d'illustration, nous avons réalisé l'expérimentation de la figure 3.18. Nous avons augmenté la durée d'exécution de la phase de préparation de la partition P7 en ajoutant 452 entrées dans la MMU ce qui a prolongé de 15 microsecondes la durée de cette phase (105 microsecondes au lieu des 90 requises).

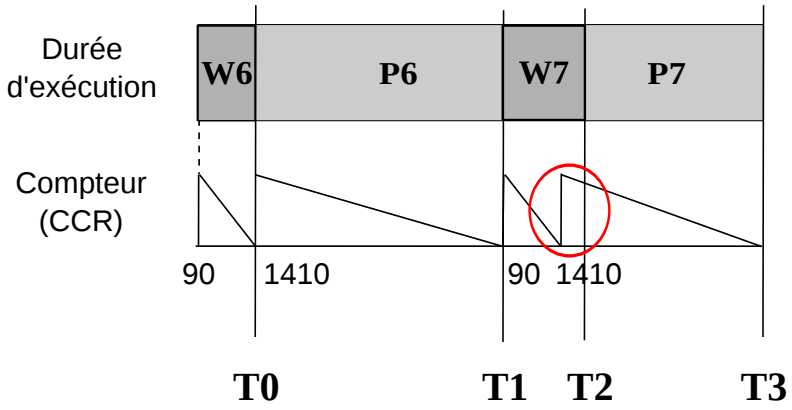

FIG. 3.18 – Dépassement de la phase de préparation de P7.

Cette expérimentation a juste pour but de montrer que l'on peut déborder de ces 90μs mais elle ne peut directement être exploitée par une partition malveillante. Si une partition malveillante utilise volontairement énormément d'entrées dans la MMU, elle va simplement se pénaliser elle-même puisque son cycle de préparation va dépasser 90μs, et par conséquent, son temps d'exécution va en être réduit d'autant. En revanche, si on imagine qu'une partition malveillante ait la possibilité de modifier la configuration d'une partition critique de façon à lui faire utiliser énormément de mémoire, il est possible qu'elle puisse donc réduire son temps d'exécution. Cependant, cette configuration étant réalisée de façon statique à priori, il semble particulièrement difficile de réaliser une telle attaque. Si l'on suppose cependant que cette configuration puisse être modifiée, alors cela peut notamment être intéressant si on le combine à l'attaque présentée dans la section précédente. C'est ce que nous montrons dans la sous-section suivante.

3.6.3 Utilisation conjointe des deux attaques

Dans ce dernier exemple (figure 3.19), nous avons cumulé les deux cas précédents. Nous considérons une partition malveillante P6 qui tente d'allonger son temps d'exécution pour voler du temps d'exécution de la partition P7, qui est non-malveillante. Par ailleurs, la partition P7 utilise énormément d'entrées dans la MMU et voit son temps de préparation déborder des 90ms. Dans ce cas très particulier, la partition P7 perd une partie conséquente de sa tranche de temps.

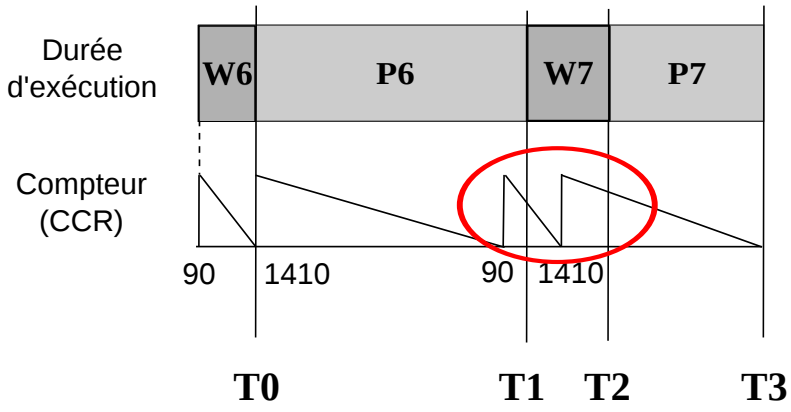

FIG. 3.19 – Dépassement de la phase de préparation et de la phase opérationnelle de P7.

3.7 Attaques ciblant les mécanismes de tolérance aux fautes

Un composant typique de tolérance aux fautes est le gestionnaire d'erreurs du noyau, utilisé comme système de détection d'erreur et système de recouvrement. Ces mécanismes peuvent provoquer un redémarrage du module quand des erreurs sont considérées comme non-récupérables. Nous avons réalisé une expérience pour tester la robustesse de ce mécanisme de gestion des erreurs en utilisant un programme *Crashme* : la partition malicieuse génère une centaine d'instructions aléatoires puis les exécute.

3.7.1 Réalisation du programme Crashme

Un programme *Crashme* consiste à générer des instructions aléatoires puis à les exécuter. Pour réaliser cela depuis une application, deux solutions sont possibles. Elles consistent à :

- modifier le code source de l'application et utiliser une zone de mémoire avec les privilèges d'écriture et d'exécution :
- modifier le fichier binaire de l'application pour exécuter les instructions aléatoires dans une zone de mémoire avec les privilèges d'exécution (donc comme n'importe quelle autre instruction de l'application).

La première solution est plus simple à réaliser que la deuxième solution mais elle nécessite des privilèges d'écriture et d'exécution sur une zone de la mémoire, ce qui qui ne devrait pas être possible justement pour éviter des exécutions de code arbitraire depuis une partition. Nous avons cependant décidé d'utiliser la première solution, puisqu'elle évite notamment de recharger l'application sur le P4080 à chaque test, même si nous considérons qu'un attaquant utiliserait plutôt la deuxième solution, qui évite de créer une zone de mémoire particulière avec des privilèges élevés (écriture et exécution). Pour notre preuve de concept, la première solution est suffisante.

La création d'un générateur de nombres pseudo-aléatoires est complexe et difficile à prouver mais notre intérêt, ici, est de générer des instructions différentes à chaque exécution de notre partition et non de générer des nombres sans aucun biais. Nous avons utilisé une formule simple et rapide, le standard minimal de Park et Miller [PM88] :

$$X_{n+1} = (16807 * X_n) \ mod \ (2^{31} - 1)$$

La première valeur X_0 est initialisée à la valeur du registre TBL (*Time Base Lower*) qui est une mesure du temps de la plateforme variant très fortement car basé sur les nanosecondes. Cette formule permet donc de générer jusqu'à 2 147 483 647 ($2^{31} - 1$) valeurs différentes, mais sur cet intervalle, nous utiliserons seulement 2^{16} valeurs (2 octets) pour générer plus simplement les 4 octets (2 fois 2 octets) composant une instruction (4 octets). Une boucle dans notre programme nous permet de générer des centaines de valeurs de 2 octets et donc des centaines d'instructions.

3.7.2 Instrumentation du noyau

Pour identifier et comparer les résultats, nous avons modifié légèrement le noyau et enregistré les erreurs générées. Nous avons également empêché le noyau de s'arrêter ou de redémarrer lorsque certaines erreurs étaient présentes. Le noyau gère 22 erreurs différentes, identifiées dans le P4080 comme des IVOR (*Interrupt Vector Offset Register*) : SRESET, *Machine Check*, DSI (*Data Storage Interrupt*), ISI (*Instruction Storage Interrupt*), ALG (*Alignment*), FPUNAV (*Floating-Point Unavailable*), DEC (*Decrementor*), PMC (*Performance Monitor Controller*), ITLB (*Instruction TLB*), DTLB (*Data TLB), TIMER,* Watchdog*, Debug*, *Processor Doorbell*, *Processor doorbell critical*, *Guest Processor Doorbell*, *Guest Processor Doorbell Critical*, *Hypersyscall*, *Hypervisor Privilege*, *Program*, *trace* et *Non-Maskable Interrupt.*

A chacune de ces erreurs, nous avons ajouté quelques lignes de code pour incrémenter un compteur. Nous utilisons une zone de la mémoire accessible en lecture et écriture pour stocker ces valeurs de compteurs. Nous avons également ajouté un compteur pour compter le nombre de cycle (un cycle = 5ms) exécutés. De cette manière, nous avons exécuté plus d'un million de cycles (environ 1h30 d'exécution) puis nous avons comparé les résultats.

3.7.3 Résultats obtenus

Ce programme Crashme a été exécuté pendant 1 043 806 cycles, ce qui nous permet de donner des statistiques fiables sur les comportements observés. Cette expérience nous a montré que parmi les 22 erreurs possibles, plus de 99,9% des erreurs sont des *Program Exceptions*. Le pourcentage restant est composé de 7 autres erreurs (*Data TLB*, *Instruction TLB*, *DSI*, *ISI*, *Alignment*, *Hypervisor syscall* et *syscall*) et il représente moins d'1% des erreurs. Les 14 autres erreurs ne sont jamais levées.

Le tableau suivant indique comment le *Health Monitor* (HM) réagit lorsqu'une erreur apparaît et montre que beaucoup d'entre elles provoquent un redémarrage ou un arrêt du module. Même si le pourcentage d'apparition de ces erreurs est très faible comparé au pourcentage de l'erreur *Program*, qui elle provoque un simple redémarrage de la partition, leur nombre est elevé. Quasiment toutes les erreurs, autres que l'erreur *Program* provoquent un redémarrage ou un arrêt du module et donc un déni de service. Cette expérience est donc intéressante au sens où elle a montré que le mécanisme de traitement des erreurs devait clairement être amélioré de façon à mieux identifier la source de l'erreur et d'agir de façon adaptée. Ces problèmes ont ainsi été présentés à Airbus et ont été pris en compte de façon à corriger les mécanismes de traitement des erreurs.

Type d'erreur	%	Red. partition	Red. Mod.	Arrêt module
Program	99,92	v	-	-
Data TLB	0,0475	v	-	-
DSI	0,0252	-	v	v
ITLB	0,0038	-	v	-
ISI	0,00086	-	-	v
Align	0,00134	-	v	-
Hypervisor syscall	0,00067	-	v	-
Syscall	0,00063	-	-	-

Conclusion

Nous avons vu dans ce chapitre de quoi était constitué la plate-forme d'expérimentation que nous avons utilisée. Nous avons également décrit plusieurs attaques réalisées sur ce système en utilisant différents canaux (gestion de la mémoire, communication, gestion du temps, mécanisme de tolérance aux fautes). Comme nous l'avons vu, plusieurs problèmes ont été identifiés, et ont été corrigés par Airbus après leurs découvertes. Ce cercle vertueux s'est ainsi montré très efficace pour améliorer la sécurité de ce noyau temps réel, d'autant plus que ces vulnérabilités, en général incluses dans du logiciel très bas niveau et liées à des mécanismes matériels propres de la plateforme, ne peuvent être détectées que grâce à ce type d'expérimentations et corrections.

Dans le prochain chapitre, nous allons aborder les mécanismes de protection que l'on peut imaginer mettre en place pour faire face à ce genre d'attaques. Nous présenterons les contre-mesures spécifiques aux vulnérabilités que nous avons découvertes sur notre noyau particulier avec nos expérimentations adaptées à ce noyau. Elles seront présentées

par type d'attaque. Ensuite, nous proposerons des contre-mesures les plus générique possibles, qui peuvent donc servir de recommandations de sécurité lors de l'implémentation d'un exécutif temps-réel désirant implémenter du partitionnement spatial et temporel. Enfin, nous présenterons une architecture sécurisée possible pour l'utilisation d'un périphérique réseau.

Chapitre 4

Protections des systèmes avioniques contre les malveillances

Introduction

Après avoir présenté dans les chapitres précédents, la classification des attaques possibles ciblant les couches basses logicielles des systèmes embarqués et après avoir mené des expérimentations mettant en œuvre de telles attaques, il est fondamental d'aborder pour terminer ce manuscrit, les mécanismes permettant de se protéger de ce type d'attaques. Nous allons donc détailler dans ce chapitre les différents mécanismes de protection que nous envisageons face à ces menaces.

Nous commençons par proposer des contre-mesures spécifiques, qui permettent de nous protéger contre les attaques que nous avons mises en œuvre sur notre plateforme d'expérimentation. Ces contre-mesures spécifiques sont abordées dans la section 4.1 et elles sont présentées par type d'attaque (suivant notre classification des attaques du chapitre 2).

Nous proposons ensuite, à partir de ces contre-mesures spécifiques, de dégager des grandes familles de mesures génériques, qui peuvent être appliquées de façon générale aux systèmes embarqués. Ces contre-mesures génériques sont abordées dans la section 4.2. Nous listons également des recommandations générales pour les développeurs de façon à éviter l'introduction de vulnérabilités lors de la phase de développement.

Enfin, nous détaillons dans la section 4.3, une proposition d'architecture sécurisée pour une carte réseau AFDX afin de se protéger au mieux contre d'éventuelles attaques logicielles. Même si cette architecture est présentée à l'aide d'un exemple précis, elle a pour but de poser les bases d'une architecture générique sécurisée de pilote de périphérique pour systèmes embarqués.

4.1 Contre-mesures spécifiques à notre plateforme d'expérimentation

Dans cette section, nous détaillons les contre-mesures adaptées à notre plateforme d'expérimentation P4080 qui permettent de se protéger contre les attaques suivantes :

- les attaques ciblant la gestion de la mémoire ;
- les attaques ciblant les communications ;
- les attaques ciblant la gestion du temps ;
- les attaques ciblant les mécanismes de tolérance aux fautes.

4.1.1 Attaques ciblant la gestion de la mémoire

Dans la section 3.4, nous avons montré qu'un accès non limité au CCSR par les partitions peut avoir de graves conséquences sur le système. Rappelons que cette zone de 16Mo permet de configurer les composants du P4080. Elle contient entre autres la configuration de certains composants, comme le *Security Engine* ou encore le *Run Control/Power Management*. Ces composants peuvent redémarrer ou stopper le système si certaines valeurs dans les registres de cette zone sont positionnées. Ce choix d'implémentation concernant les droits d'accès au CCSR s'explique de la façon suivante. Dans l'implémentation actuelle du noyau, l'accès au CCSR est autorisé à toutes les partitions pour leur permettre d'envoyer/recevoir des messages via le réseau AFDX. Plus précisément, les communications AFDX sont effectuées via un transfert DMA initialisé en configurant certains registres du CCSR. Cette solution est en fait un moyen rapide pour fournir un accès AFDX aux partitions qui en ont besoin.

La protection la plus évidente qu'il convient de préconiser consiste à apporter une modification sur les entrées de la MMU pour limiter les accès privilégiés de chaque partition au CCSR, mais cela empêche toute partition d'effectuer directement une communication AFDX. Cette solution doit donc s'accompagner de la création d'une partition dédiée aux communications AFDX. Cette partition est la seule partition (appelée "partition I/O") permettant de communiquer directement sur le réseau AFDX. Les autres partitions nécessitant des accès AFDX pourront alors avoir recours à cette partition en utilisant des mécanismes IPC (*Inter Processus Communication*). Ce choix d'architecture limite fortement les risques dûs aux malveillances concernant l'écriture dans la zone CCSR. La figure 4.1 illustre une telle implémentation. Sur ce schéma, les partitions P0 et P1 communiquent avec la partition I/O afin d'accéder à la carte AFDX. Le noyau gère toutes les autorisations. Il contrôle l'ensemble des données échangées et autorise ou non les partitions à établir des communications via l'AFDX.

En revanche, la partition I/O doit être associée à un niveau de criticité élevé. De plus, cette partition est un passage obligé pour toute requête d'entrées/sorties sur le bus AFDX et cela a un impact en terme de performance temporelle. Nous ferons une autre proposition tout aussi sécurisée mais plus performante dans la section 4.3.

Protéger efficacement la mémoire signifie aussi la protéger des accès DMA malveillants réalisés par les périphériques. Le P4080 dispose de plusieurs PAMUs (*Platform*

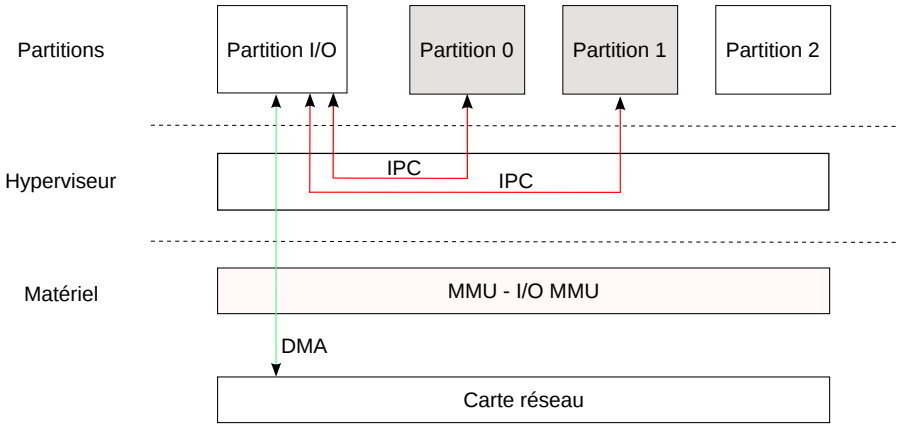

FIG. 4.1 – Exemple d'accès entre 2 partitions et une partition I/O.

Memory Management Units) qui sont identiques aux I/O MMUs (*Input/Output Memory Management Units*) présents sur les architectures Intel. Ces PAMUs permettent d'autoriser ou non les accès à la mémoire depuis les périphériques vers la mémoire mais aussi les accès de périphérique à périphérique. Cette protection se base sur l'affectation à chaque périphérique d'identifiants LIODN (*Logical I/O Device Number*) qui sont vérifiés par les PAMUs lors des accès à la mémoire. Chaque PAMU est configuré grâce à des tables, qui permettent de virtualiser la mémoire mais aussi d'attribuer des droits d'accès aux différentes zones de mémoire, de façon similaire aux tables de la MMU.

4.1.2 Attaques ciblant les communications

Plusieurs attaques ciblant les communications sont détaillées dans la section 3.5. Les contre-mesures associées sont présentées en deux parties. La première partie est consacrée aux contre-mesures pour les attaques ciblant les communications AFDX et la deuxième partie est consacrée aux contre-mesures pour les attaques ciblant les IPC.

4.1.2.1 Communications AFDX

Les communications AFDX permettent d'échanger des données entre modules via le réseau AFDX. Cependant, les mécanismes utilisés à l'heure actuelle possèdent un certain nombre de faiblesses et peuvent être corrompus (voir sous-section 3.5.1). Nous avons ainsi montré qu'il était possible depuis une partition qui utilise AFDX, de réaliser des accès DMA généralisés, de modifier des variables globales partagées entre partitions, ou encore de modifier le contenu des données de configuration qui sont envoyées au *Flash Manager*.

La première contre-mesure que nous proposons, concernant les accès DMA, consiste à protéger l'accès au CCSR. Elle nécessite l'utilisation d'une partition I/O et elle a été explicitée dans la section précédente. La deuxième contre-mesure que nous proposons

concerne l'accès à certaines variables globales depuis une partition qui utilise l'AFDX (partage qui est à l'origine d'une des attaques présentée dans le chapitre 3). Dans ce cas, nous préconisons de limiter l'accès de ces variables à la lecture uniquement. La modification de ces variables ne doit être effectuée que par le noyau, voire par la partition I/O.

La troisième contre-mesure concerne la reprogrammation de la zone EEPROM par le *Flash Manager*. Il est nécessaire d'intégrer au *Flash Manager* un mécanisme de vérification d'intégrité des données utilisées pour la reprogramation. Cependant, l'absence de ce mécanisme aujourd'hui peut se justifier par la présence en amont de la vérification de ces données avant l'envoi au *Data Loader*. En effet, les données de reprogrammation sont particulièrement critiques puisqu'elle sont utilisées pour la reconfiguration complète du calculateur. Toutefois, ajouter un mécanisme supplémentaire de contrôle d'intégrité au moment de la reprogrammation nous semble pertinent, vu la criticité des données manipulées.

4.1.2.2 Communication IPC

Nous avons également présenté des vulnérabilités dans l'implémentation des IPC, permettant à une partition d'invoquer des fonctions d'une autre partition. Il est possible de mettre en défaut ce mécanisme de différentes façons : (1) en invoquant l'appel-système de retour de la partition distante avant celui de la partition locale ; (2) en forçant le *timer* courant à s'arrêter pendant l'exécution de la fonction distante ; et (3) en modifiant le registre GPR 10 avant le premier appel-système.

La première contre-mesure consiste à empêcher la partition distante d'invoquer l'appel-système si l'appel-système de la partition locale n'a encore été invoqué. Pour cela, il est nécessaire de mémoriser et contrôler l'ordre des appel-système dans le noyau, en ajoutant des variables d'états contrôlées par le noyau. La figure 4.2 représente une implémentation possible des IPCs en utilisant un bit de vérification (noté b) et les mêmes appel-système que dans l'implémentation précédente (notés `sc`).

Le noyau débute son initialisation en positionnant à 0 un bit par invocation possible entre une partition locale et une partition distante. Dans un comportement normal, l'appel-système de la partition locale est exécuté avant l'appel-système de la partition distante. Aussi, lors de l'appel-système, le noyau passe à 1 le bit correspondant. Lorsque l'appel-système de retour de la partition distante est exécuté, le noyau vérifie si le bit correspondant est à 1. Si tel est le cas, le retour dans la partition locale est autorisé. En revanche, dans le cas d'un comportement abusif, l'appel-système de retour de la partition distante est exécuté en premier. Au moment de la vérification par le noyau, le bit correspondant étant à 0, le retour dans la partition locale n'est pas autorisé.

Ce mécanisme de vérification nécessite donc 1 bit par appel système entre chaque partition locale et chaque partition distante. L'occupation mémoire engendré est donc raisonnable et l'exécution des routines de vérification peut être réalisée en temps borné. Par conséquent, le coût de ce mécanisme de protection nous semble donc léger, bien adapté à un système temps réel et constitue donc une contre-mesure appropriée selon nous. En revanche, ce mécanisme ne prend pas en compte la possibilité d'avoir des

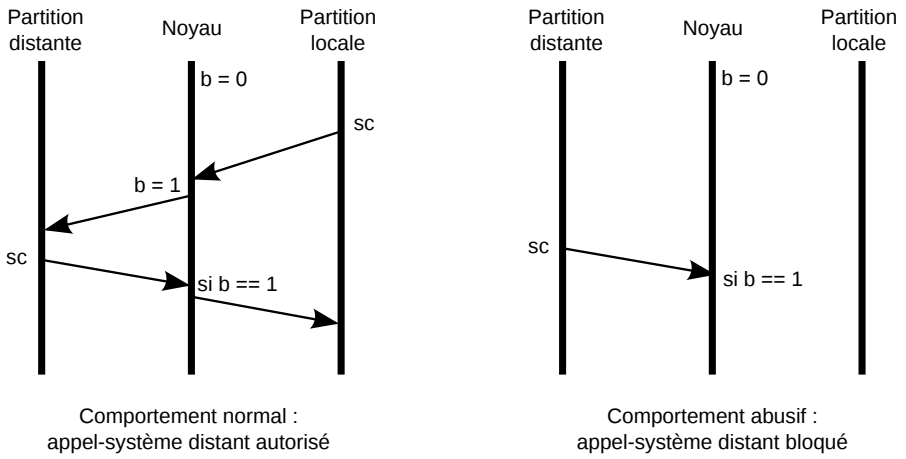

FIG. 4.2 – Proposition de contre-mesure sur le mécanisme IPC.

systèmes réentrants puisqu'un seul bit est considéré par appel-système.

La deuxième contre-mesure que nous proposons vise à éviter un blocage dans le noyau lorsque le *Global Timer* expire pendant l'exécution de la fonction distante, qui a intentionnellement exécuté une boucle infinie. Cette contre-mesure consiste à améliorer la gestion des erreurs par le *Health Monitor* du noyau. Ce dernier peut par exemple détecter que dans le cas d'une IPC, l'instruction venant d'être exécutée provient de la partition distante (en fonction de l'adresse de cette instruction) et non de la partition locale. Il convient donc, soit de revenir dans la partition locale après l'appel à la fonction distante, avec un code d'erreur, soit d'arrêter la partition locale pour ne plus réaliser d'appel à la fonction.

La troisième contre-mesure concerne le registre `GPR10` qui est utilisé dans le code de la partition locale avant l'appel-système et qui est ensuite vérifié par le noyau. Ce registre étant un registre général sans privilège particulier, la partition peut modifier cette valeur. Ce registre devrait donc être un registre privilégié modifiable seulement par le noyau.

4.1.3 Attaques ciblant la gestion du temps

Nous avons présenté dans la section 3.6 des attaques ciblant la gestion du temps. Celles-ci peuvent permettre à une partition malveillante d'allonger sa durée d'exécution, ce qui peut avoir pour conséquence de limiter la durée d'exécution de la partition suivante (en particulier lors de la phase de préparation de la partition suivante). Pour réaliser ce débordement, il suffit de provoquer une interruption à la fin de la tranche de temps de la partition malveillante. Cette interruption est traitée par le noyau, qui est non-interruptible. Par conséquent, l'ordonnancement des partitions est retardé. Il semble particulièrement difficile d'imaginer une contre-mesure technique à cette attaque. En revanche, on peut imaginer une contre-mesure concernant le calcul des WCET : il convient

de prendre en compte la durée d'exécution de l'interruption la plus longue dans le calcul des différents WCET.

Nous avons également montré qu'il est possible d'allonger la phase de préparation en utilisant un nombre très élevé d'entrées dans la MMU (pour utiliser un grand nombre de zones de mémoire différentes). La configuration de la MMU étant réalisée par le noyau, ce code est encore une fois non-interruptible, ce qui signifie qu'il est susceptible de poursuivre son exécution alors qu'un *timer* est en attente et que ce *timer* ne pourra déclencher une interruption que lorsque le processeur reprendra l'exécution de la boucle infinie en mode utilisateur. La contre-mesure que nous proposons ici est de limiter le nombre d'entrées dans la MMU directement par le générateur de code. Ceci empêche la création de trop nombreuses entrées sachant qu'il est très peu probable que toutes ces entrées soient réellement nécessaires. Il faut également que le temps de préparation de la partition prenne en compte le temps nécessaire à la configuration de toutes les entrées de la MMU.

La figure 4.3 illustre les durées d'exécutions des partitions en prenant en compte les durées d'exécution des interruptions.

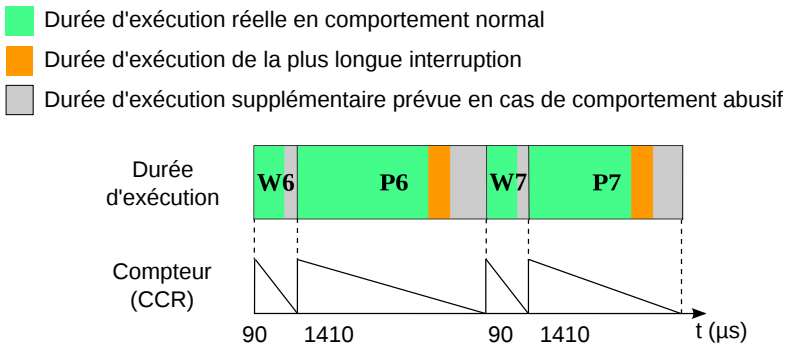

FIG. 4.3 – Proposition de contre-mesure prenant en compte les durées d'exécution d'interruption.

Sur la figure 4.3, nous présentons 2 partitions P6 et P7 avec leurs phases de préparation respectives W6 et W7. Les durées P6 et P7 sont constituées chacune de trois durées d'exécution : la durée d'exécution de la partition elle-même, la durée d'exécution de l'interruption la plus longue et enfin la marge de sûreté habituelle (de l'ordre de 20%). Pour les durées W6 et W7, il suffit de limiter le nombre d'entrées dans la MMU à une cinquantaine au maximum au lieu de 512 permises par défaut par la MMU.

4.1.4 Attaques ciblant les mécanismes de tolérance aux fautes

Nous avons présenté dans la sous-section 3.7.1 la réalisation d'un programme *Crashme* qui nous a permis de détecter plusieurs dysfonctionnements dans le mécanisme de gestion des erreurs. Il est ainsi possible de redémarrer ou arrêter le module complet depuis une partition. Nous proposons de modifier le comportement du gestionnaire d'erreur pour

ne pas autoriser de redémarrages intempestifs ou d'arrêt du module. Lorsqu'une erreur survient, le noyau doit sanctionner la partition à l'origine de l'erreur uniquement et non le module. Il doit donc établir un diagnostic de fautes très précis.

La contre-mesure consiste à mieux identifier l'origine de l'erreur, ce qui est possible puisque le type de l'erreur est connu par le noyau ainsi que la partition concernée (le contexte d'exécution au moment du déclenchement de l'erreur). De cette façon, il a été possible de prendre des sanctions précises concernant uniquement l'origine de l'erreur et non plus des sanctions trop larges appliquées souvent à tout le module. Cette modification a été implémentée dans le noyau après nos expérimentations.

Plus précisément, voici quelques éléments techniques qui ont été utilisés pour améliorer cette gestion des erreurs. Pour ce qui concerne l'origine de l'erreur, le cœur e500mc du P4080 comporte un mécanisme commun aux processeurs PowerPC : lorsqu'une interruption survient, le registre `SRR0` contient la valeur de l'adresse qui a été interrompue[41] et le registre `SRR1` contient la valeur du registre `MSR` (*Machine Status Register*) qui contient le niveau de privilège actuel. Ces deux registres permettent notamment de se "repositionner" à l'instruction suivante avec les mêmes privilèges. Si une partition est la source de l'erreur, alors il est nécessaire d'arrêter le fonctionnement de cette partition. Si c'est le noyau qui en est la source, alors il est préférable de redémarrer ou d'arrêter le module complet car des défaillances à répétitions peuvent se produire. En ce qui concerne la nature de l'erreur, le registre ESR (*Exception Syndrome Register*) permet d'informer le noyau de la cause exacte de l'erreur (instruction illégale, instruction privilégiée, opération `store`, etc.).

Le tableau 4.1 présente les résultats avec le nouveau gestionnaire d'erreurs.

Type d'erreur	%	Redémarrage de la partition	Redémarrage du module	Arrêt du module
Program	99,92	v	-	-
Data TLB	0,0475	v	-	-
Data Storage Int.	0,0252	v	-	-
Instruction TLB	0,0038	v	-	-
Instruction Storage Int.	0,00086	v	-	-
Alignment	0,00134	v	-	-
Hypervisor syscall	0,00067	-	-	-
Syscall	0,00063	-	-	-

TAB. 4.1 – Répartition des actions en fonction des types d'erreurs.

Les taux d'erreurs sont identiques à l'ancienne version mais il n'est désormais plus possible de faire redémarrer ou arrêter le module depuis un code utilisateur d'une partition car la sanction se fait uniquement au niveau de la partition. En ce qui concerne

[41] Sauf en cas d'erreur imprécise.

l'appel-système hyperviseur (*hypervisor syscall*), il ne fait pas redémarrer la partition, mais simplement passer à l'instruction suivante. Ce choix a été fait car cet appel système n'est pas une erreur en tant que tel, puisqu'il s'agit d'une instruction parfaitement légitime. Cependant, il est a priori improbable qu'elle soit invoquée dans notre système expérimental puisqu'aucun système d'exploitation invité n'y est présent.

Dans cette section, nous nous sommes focalisés sur les contre-mesures spécifiques à notre plateforme d'expérimentation et aux attaques que nous avons pu mener. Il est possible de généraliser certaines de ces contre-mesures de façon à proposer des mécanismes de protection génériques, adaptés à différents types de systèmes embarqués. C'est l'objet de la section suivante.

4.2 Contre-mesures génériques

Dans cette section, nous commençons par proposer des contre-mesures génériques adaptables à différents systèmes embarqués, face aux attaques suivantes :

- les attaques ciblant le processeur ;
- les attaques ciblant la gestion de la mémoire ;
- les attaques ciblant les communications ;
- les attaques ciblant la cryptographie ;
- les attaques ciblant les fonctions ancillaires ;
- les attaques ciblant les mécanismes de tolérance aux fautes.

Ensuite, nous proposons de compléter ces contre-mesures par des recommandations aux développeurs (voir section 4.2.8). En particulier, nous présentons brièvement les techniques d'analyse statique (voir sous-section 4.2.9), qui permettent de vérifier le code source de l'applications et des techniques d'analyse dynamique (voir sous-section 4.2.10) qui permettent d'analyser l'exécution en temps-réel de l'application.

4.2.1 Attaques ciblant le processeur

Le rôle central du processeur dans le système en fait un composant souvent ciblé par les attaques. Effectivement, la compromission du processeur permet soit d'exécuter un code malveillant, soit de porter atteinte à l'intégrité des périphériques. La plupart des contre-mesures présentées dans cette sous-section visent à consolider les configurations des différents éléments qui composent le processeur (cache, mécanisme de synchronisation, gestionnaire des interruptions, etc.).

Les accès aux périphériques et à la mémoire centrale étant plus lents que la vitesse de fonctionnement du processeur, des mémoires caches avec des vitesses de fonctionnement intermédiaires sont utilisées comme espace de stockage temporaire. Ces espaces de stockage sont partagés entre les différents processus [42] qui s'exécutent sur le processeur. Il est important que, lors du changement de contexte, le contenu de ces caches soit complètement vidé. Cette contre-mesure assure qu'un processus malveillant ne puisse pas

[42] Le terme processus est utilisé au sens large et fait référence à une tâche, une partition, etc.

stocker dans une des mémoires caches des instructions machines en espérant que le processeur les exécutera en pensant qu'elles correspondent aux instructions machines d'un autre processus. De plus, si la propriété de confidentialité est importante à assurer dans le système, le vidage du contenu des mémoires caches doit être réalisé indépendamment du contenu de ces mémoires.

Les processeurs modernes sont également structurés en plusieurs cœurs pour l'exécution en parallèle des instructions. Sur la plupart des architectures, chaque cœur dispose de son propre gestionnaire d'interruptions (nommé *Local APIC* sur les architectures Intel). La synchronisation entre ces cœurs est également assurée via ces gestionnaires d'interruptions. La configuration de ces gestionnaires doit être réalisée lors de l'initialisation du noyau du système et elle doit être accessible uniquement par ce noyau. De la sorte, un processus ne peut pas envoyer des interruptions pour, par exemple, arrêter un des autres cœurs.

Tous les cœurs du processeur doivent être initialisés. Cette initialisation peut être mise en place par une chaîne de confiance au démarrage. A titre d'exemple, pour la mise en place d'une chaîne de confiance, le processeur P4080 peut utiliser le composant *Security Monitor* et les processeurs Intel peuvent utiliser le composant TPM.

Le processeur doit pouvoir exécuter des instructions machines privilégiées (telles que les instructions de manipulation des registres de configuration) uniquement si le niveau de privilège du processeur est lui-même élevé. Pour assurer ce contrôle, deux principaux mécanismes de protection doivent être configurés : les anneaux de protection (*ring*) avec la segmentation et les droits d'accès à la mémoire avec la pagination. L'anneau le plus privilégié doit correspondre aux instructions du noyau et toutes les instructions du noyau doivent être associées à cet anneau. Quant à la pagination, elle permet de protéger ces instructions de manière à empêcher un processus malveillant de les modifier dans le but de faire exécuter une séquence d'instructions malveillante au processeur.

Les processeurs modernes disposent de plusieurs modes de fonctionnement (par exemple, pour la gestion des ressources, pour la virtualisation, etc.). Les processeurs du grand public intègrent également des modes historiques pour des raisons de rétrocompatibilités pour les logiciels. Il est important que le noyau configure tous ces modes. Les modes qui ne sont pas utilisés par le système doivent aussi être configurés et figés. Cette contre-mesure empêche des attaques de profiter des modes du processeur qui ont été ignorés par le noyau pour exécuter des actions en contournant les mécanismes de protection des modes effectivement configurés.

Plus généralement, une contre-mesure importante est l'analyse du point de vue de la sécurité de la spécification du processeur, lorsque cette spécification est connue du développeur du système. Cette spécification peut correspondre au code VHDL utilisé pour la conception. Par exemple, le code VHDL du processeur Leon3 est disponible pour permettre aux développeurs d'y intégrer des modifications. Avoir à disposition la spécification d'un processeur permet d'identifier des vulnérabilités (généralement des instructions qui ne sont pas – ou mal – documentées). Par la suite, l'idéal est de modifier la spécification du processeur pour éliminer ces vulnérabilités. Si cette option n'est pas envisageable, ces vulnérabilités doivent alors être contenues par des mécanismes de pro-

tection adaptés et intégrés dans le noyau du système. Lorsque cette spécification n'est pas connue du développeur du système, une analyse de vulnérabilités doit tout de même être réalisée en utilisant des outils de *fuzzing*. Le programme *Crashme* en est un bon exemple. Les vulnérabilités identifiées par ces outils doivent être également contenues par des contre-mesures à intégrer dans le noyau du système.

4.2.2 Attaques ciblant la mémoire

La mémoire d'un système correspond à l'ensemble des mémoires accessibles depuis le processeur. Il s'agit de la mémoire centrale (habituellement nommée RAM) et de la mémoire des périphériques (par exemple, les registres des périphériques PCI-Express). Ces mémoires peuvent être directement manipulées par le processeur mais elles peuvent aussi être manipulées via des transferts DMA (*Direct Memory Access*). Par exemple, dans le cas d'un transfert DMA, un périphérique peut recopier le contenu d'un tableau en mémoire centrale sans passer par le processeur. Ces mémoires sont donc manipulées par différents processus et périphériques de différents niveaux de criticité. Dans la suite, le terme composant fait référence à un processus ou un périphérique. Une zone mémoire peut être attribuée de manière exclusive à un composant ou peut être partagée par plusieurs composants. Pour empêcher un composant d'interférer avec l'espace mémoire attribué et manipulé par un autre composant, il est nécessaire de mettre en place des contre-mesures.

Le premier accès réalisé à une zone de la RAM attribuée de manière exclusive à un processus doit toujours être une écriture. Autrement dit, l'initialisation d'un processus doit toujours débuter par une écriture dans la zone mémoire pour placer l'environnement d'exécution du processus dans un état déterministe. De la même manière, dans le cas de l'allocation dynamique ou de l'attribution temporaire d'une zone mémoire par le noyau, le processus doit avant toute manipulation de ces nouvelles zones mémoires, placer leur contenu dans un état déterministe. Ces bonnes pratiques empêche un processus de copier un contenu dans une zone mémoire de manière à interférer avec un autre processus qui récupèrera un accès à cette zone mémoire.

Les mécanismes de pagination et de segmentation de la MMU (*Memory Management Unit*) doivent être mis en place par le noyau pour assurer le partitionnement spatial de la mémoire des processus. Ces mécanismes permettent tout d'abord de virtualiser la mémoire et ensuite d'indiquer quels sont les droits d'accès à la mémoire (lecture, écriture ou exécution). La mise en place de la mémoire virtuelle consiste à construire une fonction de traduction entre une adresse virtuelle (manipulée par le processus) et une adresse physique. La granularité de cette fonction est la page mémoire (il s'agit d'une unité correspondant à 4ko). Il existe donc les pages mémoires virtuelles et les pages mémoires physiques. Les droits d'accès aux pages mémoires sont indiqués directement dans la fonction de traduction. Il est important de noter qu'il est possible de construire des fonctions différentes pour les différents processus. Lors de la construction de chaque fonction, seules les pages mémoires physiques contenant des données d'un processus doivent avoir une page mémoire virtuelle comme image. Aussi, pour faciliter la mise en place de ces fonctions, le noyau doit charger les processus de manière à ne pas partager une page

mémoire physique entre plusieurs processus. Dans le cas de la mémoire partagée, une page mémoire contenant uniquement les données partagées entre les processus doit être créée. Ce mécanisme de protection met en place le partitionnement spatial pour les accès mémoire depuis le processeur. Il doit s'accompagner d'un mécanisme complémentaire pour les accès réalisés avec les périphériques.

De la même manière que précédemment, une fonction de traduction peut être mise en place pour les accès entre les périphériques et la mémoire centrale. Ces fonctions sont construites en configurant une MMU pour les entrées-sorties (I/O MMU sur les architectures Intel et PAMU sur le P4080). Si un périphérique est partagé entre plusieurs processus et que la virtualisation du matériel n'est pas disponible (cf. section 4.3), il est alors plus pertinent de construire un processus dédié aux communications avec le périphérique : un pilote de périphérique. Les autres processus peuvent alors invoquer les services de ce pilote de périphérique. Tout comme pour la configuration de la MMU précédente, les fonctions doivent uniquement permettre à un périphérique d'accéder aux pages de la mémoire physique qui le concerne. De la sorte, le système est prémuni des attaques via les transferts DMA.

4.2.3 Attaques ciblant les communications

Les mécanismes de communications permettent à différents processus d'échanger de l'information. Ces échanges doivent être contrôlés de manière à s'assurer qu'un processus peut toujours envoyer ses données et qu'un processus peut recevoir des données. Autrement dit, il ne faut pas qu'un processus puisse empêcher un autre processus d'envoyer des données ou qu'un processus puisse recevoir les données destinées à un autre processus. Des mécanismes de protection doivent être mis en place pour garantir que ces propriétés sont bien assurées.

Pour permettre aux mécanismes de protection de contrôler les messages échangés, il est nécessaire que les mécanismes de communication associent aux messages les identités de la source et de la destination. Ces informations ne doivent pas être fixées par la source elle-même. Elles doivent être fixées par un composant associé au plus haut niveau de confiance et incontournable pour l'échange de messages. Ce composant peut être le noyau qui peut être sollicité par des appels-système pour permettre les échanges. Le verdict associé à la tentative d'envoi d'un message doit se baser sur une liste de contrôle des accès, idéalement construite statiquement lors de la phase de conception et de compilation des différents composants du système.

Quel que soit le type de communication mis en place entre les processus (communication asynchrone ou synchrone), un processus ne doit pas pouvoir submerger le récepteur au point de l'empêcher d'échanger à son tour des informations avec d'autres processus. Pour éviter cette situation, chaque processus doit disposer d'autant de boîtes aux lettres que de processus avec lesquels il peut communiquer. Le noyau peut alors poster le message dans la boîte aux lettres du destinataire correspondant au processus source. Ainsi, un débordement d'une boîte aux lettres issu d'une activité malveillante nuit uniquement au processus source.

Le message échangé entre les processus doit pouvoir être analysé entièrement par le

noyau. Le processus ne doit pas pouvoir utiliser un pointeur vers sa mémoire virtuelle pour indiquer au noyau le contenu du message. De plus, il est préférable que le message à échanger entre deux processus soit contenu dans une page mémoire dédiée. Ainsi, si le noyau associe cette page mémoire à la mémoire virtuelle du processus destinataire (à la manière des *blackboards*), ce dernier n'aura accès qu'au message.

4.2.4 Attaques ciblant la gestion du temps

Les contre-mesures aux attaques ciblant la gestion du temps correspondent aux mécanismes permettant de protéger la politique d'ordonnancement. Ces contre-mesures doivent donc être directement intégrées dans le noyau et elles entraînent nécessairement l'exécution de routines supplémentaires. La première contre-mesure correspond à la gestion de la sauvegarde du contexte d'un processus. L'espace mémoire associé doit être entièrement géré par le noyau et ne doit pas être accessible au processus. Par exemple sur les architectures Intel, en adoptant ce principe, un processus ne peut pas modifier le contenu du registre `cr3` (il s'agit d'un registre permettant de contrôler la traduction entre une adresse virtuelle et une adresse physique).

La seconde contre-mesure concerne le calcul du temps d'exécution d'un processus durant un cycle de l'ordonnanceur. Ce temps (le WCET) doit prendre en compte non seulement la durée d'exécution des instructions du processus mais également la durée d'exécution des routines permettant de configurer le matériel. Plus précisément, il est nécessaire de considérer la durée nécessaire pour re-configurer les composants matériels (MMU, périphériques, caches) et la durée d'exécution de l'interruption la plus longue.

4.2.5 Attaques ciblant la cryptographie

Les attaques ciblant la cryptographie visent généralement à obtenir les clés privées incluses dans les fonctions cryptographiques voire à interférer avec les méthodes permettant de générer des nombres aléatoires. Pour empêcher un processus d'obtenir ces clés, il est nécessaire d'éviter les fuites d'information en vidant systématiquement les caches du processeur lors des changements de contexte. De cette manière, un processus malveillant ne peut pas se baser sur le contenu du cache pour déduire des informations sur les clés manipulées par le processeur.

Un processus ne doit pas pouvoir deviner (totalement ou partiellement) la valeur d'un nombre aléatoire obtenu par un autre processus. Le service permettant de délivrer ces valeurs doit donc être implémenté dans un espace de confiance qui peut être un processus dédié ou le noyau. Ce service doit également, dans la mesure du possible, se baser sur un générateur aléatoire matériel pour empêcher les autres processus de surcharger le système dans le but de restreindre l'intervalle des valeurs que le service peut retourner.

4.2.6 Attaques ciblant les fonctions ancillaires

Rappelons que les fonctions ancillaires correspondent à la gestion de l'alimentation, de l'*over-clocking*, du contrôle de température, etc. Sur les architectures x86, ces fonc-

tions sont assurées dans un mode dédié du processeur : le mode SMM. Ce mode dispose d'un accès privilégié au système (dans la limite des quatre premiers giga-octets de mémoire). Il a été la cible de plusieurs attaques. Pour éviter ces attaques, les fondeurs de processeurs x86 proposent un mécanisme permettant de verrouiller la mémoire contenant les instructions exécutées par ce mode. Il est important que ce mode soit configuré et verrouillé.

4.2.7 Attaques ciblant les mécanismes de tolérance aux fautes

Comme nous l'avons montré dans la section 2.2, les attaques ciblant les mécanismes de tolérance aux fautes peuvent cibler le traitement d'erreur et le traitement des fautes. Nous allons proposer quelques contre-mesures génériques permettant de se défendre d'attaques ciblant ces mécanismes. Nous ne serons pas exhaustifs dans cette partie, puisque ce sont les attaques que nous avons pu le moins expérimenter, étant donné que le système expérimental dont nous disposions, contenait peu de mécanismes de tolérance aux fautes (il aurait fallu que nous disposions de plusieurs versions du calculateur pour utiliser la redondance par exemple, ce qui n'a pas été possible dans notre plateforme).

4.2.7.1 Traitement des fautes

Concernant le traitement de fautes, on peut généraliser la contre-mesure mise en place dans le cas spécifique de notre expérimentation. Il est fondamental que le diagnostic de fautes soit le plus précis possible, de façon à identifier avec certitude l'origine des erreurs et donc le composant fautif. Cette contre-mesure, qui a été mis en place dans notre expérimentation et qui a permis d'améliorer la gestion des erreurs, a pour vocation d'empêcher une application malveillante de déclencher volontairement de multiples erreurs dans l'espoir de provoquer un redémarrage complet du système.

La reconfiguration fait également partie du traitement de fautes. Cette reconfiguration intervient en général après identification et passivation d'un ou plusieurs composants fautifs et consiste à modifier la configuration du système pour qu'il puisse continuer à fonctionner (éventuellement en mode dégradé) sans faute. Elle est suivie d'une phase de réinitialisation dans laquelle la nouvelle configuration est enregistrée dans le système. Un attaquant ciblant la reconfiguration et réinitialisation peut tenter d'accéder à la zone mémoire où se situe cette configuration de façon à faire redémarrer le système dans une configuration incorrecte. Une contre-mesure face à ce type d'attaques est de choisir avec précaution la zone de mémoire dans le système où est stockée cette configuration. Cela fait intervenir une nouvelle fois la configuration précise de la MMU.

4.2.7.2 Traitement d'erreurs

En ce qui concerne les différents mécanismes de traitement d'erreurs, on peut tout d'abord envisager des contre-mesures visant à se protéger d'attaques ciblant la détection d'erreurs. Par exemple, la duplication et la comparaison est un mécanisme efficace de tolérance aux fautes accidentelles mais il est relativement facile à contourner par une

application malveillante. En revanche, la diversification fonctionnelle est un mécanisme efficace de détection d'erreur, même vis-à-vis des fautes intentionnelles. Il est donc à privilégier lorsque l'on s'intéresse aux malveillances. Il est en effet très difficile pour un attaquant, d'imaginer une attaque efficace qui puisse avoir les mêmes effets sur des matériels et logiciels diversifiés. Cette diversification a d'ailleurs déjà fait ses preuves vis-à-vis des malveillances dans d'autres contextes, comme dans le cadre d'opérations de maintenance des futurs architectures avions [Laa09], ou dans le cadre d'applications Web par exemple [Sai05].

En ce qui concerne les mécanismes de recouvrement d'erreurs et plus précisément la reprise, il nécessite, comme nous l'avons vu dans les précédents chapitres, l'existence de points de reprise (aussi appelés points de restauration). De façon à éviter qu'un attaquant puisse modifier ces points de reprise, une contre-mesure consiste à utiliser une zone de mémoire protégée pour stocker les points de reprise, de façon à que cette zone ne soit pas modifiable par les différentes partitions. Cela signifie qu'ils devront probablement être stockés dans l'espace du noyau.

En ce qui concerne la corruption potentielle des mécanismes de poursuite, qui provoquerait volontairement le passage en mode dégradé du système, une contre-mesure possible consiste à disposer d'un mécanisme de diagnostic de fautes très précis. Si le diagnostic de fautes évalue avec certitude l'origine d'erreurs à répétition (provoquées dans le but de passer le système en mode dégradé), il peut alors sanctionner l'application (en l'occurrence malveillante) et non le système complet.

4.2.8 Recommandation aux développeurs

Les contre-mesures proposées dans les sous-sections précédentes visent à intégrer des mécanismes de protection architecturaux. Il est bien sûr nécessaire de compléter ces mécanismes par des bonnes pratiques de développement, permettant d'éviter l'introduction de vulnérabilités dans le logiciel ou de détecter leur présence. Ces bonnes pratiques sont déjà bien sûr préconisées pour détecter les fautes accidentelles (*bugs*) mais elles sont également fondamentales pour détecter les fautes intentionnelles.

Notre objectif dans cette section est simplement de lister les bonnes pratiques de développement qui nous semblent les plus importantes vis-à-vis de l'identification de vulnérabilités. Deux d'entre elles sont notamment l'analyse statique et l'analyse dynamique de code. Elles sont l'objet de cette sous-section. Cette sous-section ne prétend pas faire une étude exhaustive de ces analyses (ce qui nécessiterait un chapitre complet) mais plutôt de souligner l'importance de leur utilisation lors du développement de systèmes embarqués.

4.2.9 Analyse statique

L'analyse statique vise à détecter dans le code source du programme cible d'éventuelles fautes. L'identification de ces fautes est utile à la fois du point de vue de la tolérance aux fautes mais aussi du point de vue de la sécurité. En effet, leur identification durant la phase de développement permet de réduire de façon considérable les fautes

de conception et ainsi de réduire les vulnérabilités exploitables dans le code source. Plusieurs classes de défaut sont identifiées par les outils d'analyse statiques [CE09]. Parmi celles-ci, nous pouvons citer les **divisions par zéro**, les **fuites de mémoire**, les **déréférencement de pointeur nul**, les **variables non-initialisées**, les **débordements de tampons**, et les **typages (*cast*) inappropriés**. Elles correspondent aux fautes les plus couramment rencontrées dans les codes.

Certains outils d'analyse statique de code sont capables de détecter la majorité de ces classes de défaut mais ils sont aussi capables de vérifier la conformité du code vis-à-vis de certains standards de développement. A titre d'exemple, Prevent[43], Polyspace[44] ou Astrée[45], peuvent effectuer ces vérifications avec des langages comme C, C++, Java ou Ada. *MALPAS Software Static Analysis Toolset*[46] effectue des vérifications similaires pour le code assembleur PowerPC.

En particulier, pour notre étude, l'analyse statique peut être réalisée sur le code source du noyau ainsi que sur les codes source des partitions, qu'ils soient écrits en C ou en assembleur PowerPC. Nous pouvons également vérifier et analyser les fichiers de configuration utilisés pour paramétrer le noyau. En effet, ces fichiers de configuration contiennent des informations sur les zones de mémoires utilisées, les tranches de temps allouées aux partitions ou encore la gestion des erreurs (*Health Monitor*). Nous avons d'ailleurs régulièrement utilisé l'analyse statique de code dans le cadre de nos expérimentations sur la plateforme P4080 et le noyau expérimental d'Airbus dont le code source nous a été fourni (ce qui nous a donc permis de réaliser ce type d'analyse). Même si nous n'avons pas systématiquement utilisé d'outils automatiques, ce type d'analyse nous a permis d'identifier des vulnérabilités intéressantes, en ce qui concerne la gestion des IPC par exemple et s'est donc révélé particulièrement efficace.

4.2.10 Analyse dynamique

L'analyse statique peut être complétée par l'analyse dynamique dont l'objectif est également l'identification de vulnérabilités mais en se basant sur l'observation du comportement du programme durant son exécution.

Des outils comme Valgrind[47] et mpatrol[48] sont des exemples de logiciels permettant de détecter des fuites de mémoire en se basant par exemple sur l'observation des invocations à des bibliothèques standards telle que la `libc`.

Une technique courante consiste à analyser le comportement d'un programme en l'exécutant en premier lieu dans un environnement restreint, dans lequel chaque accès à la mémoire et chaque entrée/sortie est contrôlé. Cet environnement peut prendre différentes formes, par exemple les *sandbox* Java ou les *Jails* des systèmes BSD. Cela peut aller jusqu'à l'utilisation des machines virtuelles complètes. Ceci est notamment couramment

[43] `http://www.coverity.com`
[44] `http://www.mathworks.fr/products/polyspace/`
[45] `http://astree.ens.fr`
[46] `http://www.malpas-global.com`
[47] `http://valgrind.org`
[48] `http://mpatrol.sourceforge.net/`

utilisé par les éditeurs d'anti-virus, qui analysent en détail toute nouvelle génération de code malveillant (en particulier auto-reproducteurs) dans des machines virtuelles dédiées, de façon à confiner l'exécution de ce type de maliciels.

Dans le cadre de nos expérimentations sur la plateforme P4080, nous avons régulièrement utilisé l'analyse dynamique, notamment pour vérifier s'il était possible de rompre l'isolation temporelle des partitions et pour analyser le comportement du noyau ou des différentes partitions critiques en présence d'une partition malveillante qui tente d'injecter des erreurs. L'analyse dynamique nous a permis d'identifier de nombreux problèmes de déni-de-service notamment.

4.3 Proposition d'une architecture sécurisée pour la gestion d'un périphérique partagé

Comme nous l'avons vu dans cette thèse, plusieurs attaques sont liées à la volonté de partager un périphérique, en l'occurrence la carte AFDX dans notre plateforme d'expérimentations, entre partitions de différents niveaux de criticité. Ce partage est réalisé par deux mécanismes : échange d'informations avec le périphérique via des accès DMA et partage du pilote logiciel du périphérique entre les partitions. En particulier, dans la version préliminaire du noyau d'Airbus, le partage du pilote logiciel est réalisé via la création d'une mémoire partagée entre les différentes partitions et le pilote. Il s'agit d'une solution temporaire pour laquelle nous proposons une amélioration dans cette section. Cette solution permet d'implémenter des mécanismes de protection efficaces vis-à-vis des accès aux périphériques partagés entre partitions et est basée sur une architecture matérielle particulière. Cette proposition d'architecture n'a pas été expérimentée dans le cadre de cette thèse mais elle a toutefois été discutée en collaboration avec des ingénieurs d'Airbus.

Nous présentons tout d'abord le principe général de cette solution, basée sur des technologies matérielles de virtualisation. Cette technologie est aujourd'hui implémentée dans un certain nombre de cartes réseaux (de type Ethernet) mais pas à notre connaissance pour des cartes AFDX. Nous présentons cette technologie, baptisée *Single Root I/O* (technologie qui n'a donc pas été spécifiquement développée pour des applications embarquées), dans la première partie de cette section. Nous présentons ensuite son utilisation dans un système avionique embarqué, utilisant le principe du partitionnement spatial et temporel.

4.3.1 Principe

L'architecture sécurisée que nous proposons est basée sur des technologies matérielles d'assistance à la virtualisation. Ces technologies sont déjà relativement matures et utilisées sur les processeurs récents, et favorisent grandement le développement de gestionnaires de machines virtuelles (aussi appelés hyperviseurs). Elles portent respectivement le nom d'Intel-VTx ou AMD-V pour les processeurs des familles Intel et AMD. Elles sont par ailleurs déjà mises à profit dans plusieurs études au service de la sécurité

des systèmes informatiques. Plusieurs hyperviseurs de sécurité ont ainsi été proposés dans différents travaux [M+10, LD11, S+09a] et ont souvent pour but d'assurer une protection efficace vis-à-vis de corruptions des couches basses du logiciel. Ainsi, souvent, les hyperviseurs de sécurité visent à protéger le noyau du système d'exploitation lui-même, qui en principe, est le logiciel le plus privilégié, mais aussi le plus sensible d'un système informatique. L'idée principale derrière ces travaux réside dans le fait qu'il est très difficile de corrompre un hyperviseur utilisant des technologies d'assistance matérielle à la virtualisation puisque ces technologies sont considérées comme très difficiles à contourner, contrairement à des mécanismes de protection logiciels.

L'architecture que nous proposons ici utilise également des techniques de virtualisation, mais au niveau des périphériques cette fois-ci. Ces mécanismes de virtualisation fonctionnent en coopération avec les mécanismes de virtualisation du processeur principal de la machine sur laquelle est connectée le périphérique. Ces mécanismes matériels incontournables et cette coopération sont selon nous une base solide pour assurer la sécurité d'un périphérique. Elle est encore très peu déployée aujourd'hui et nous semble un atout important qu'il est utile de considérer dès aujourd'hui pour l'amélioration de la sécurisation des systèmes embarqués avioniques.

4.3.2 Virtualisation matérielle d'un périphérique : *Single Root-I/O Virtualization*

Un certain nombre de périphériques réseau implémentent aujourd'hui une technologie de virtualisation matérielle baptisée *Single Root-I/O Virtualization* (SR-IOV). La carte Intel 82599EB 10Gb [49], qui peut être connectée sur un port PCIe, en est un exemple.

La virtualisation SR-IOV permet, à partir d'une unique carte physique, d'exposer de multiple cartes virtuelles, pouvant être mises à disposition de différentes machines virtuelles. La virtualisation SR-IOV est plus précisément une extension de la spécification du protocole PCIe qui permet à différentes machines virtuelles de partager les mêmes ressources physiques PCIe.

La figure 4.4 représente cette architecture.

Les accès à ces ressources physiques sont donc virtualisés au travers de l'invocation de différentes *fonctions* PCIe :

- Une fonction physique (PF) correspond à l'exposition sur le bus PCI-Express du périphérique réel ainsi que la totalité de ses ressources ; cette fonction est destinée à être accessible uniquement par du logiciel privilégié (typiquement l'hyperviseur s'exécutant sur le processeur principal ou à une machine virtuelle privilégiée de type *Dom0* de Xen par exemple) mais non accessible aux machines virtuelles ;
- Une ou plusieurs fonctions virtuelles (VF) exposent à une machine virtuelle une carte virtuelle qui correspond à un sous-ensemble de ressources de la carte physique.

A chaque PF et VF, est associé un identifiant *Express Request ID* ou RID unique,

[49] `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`

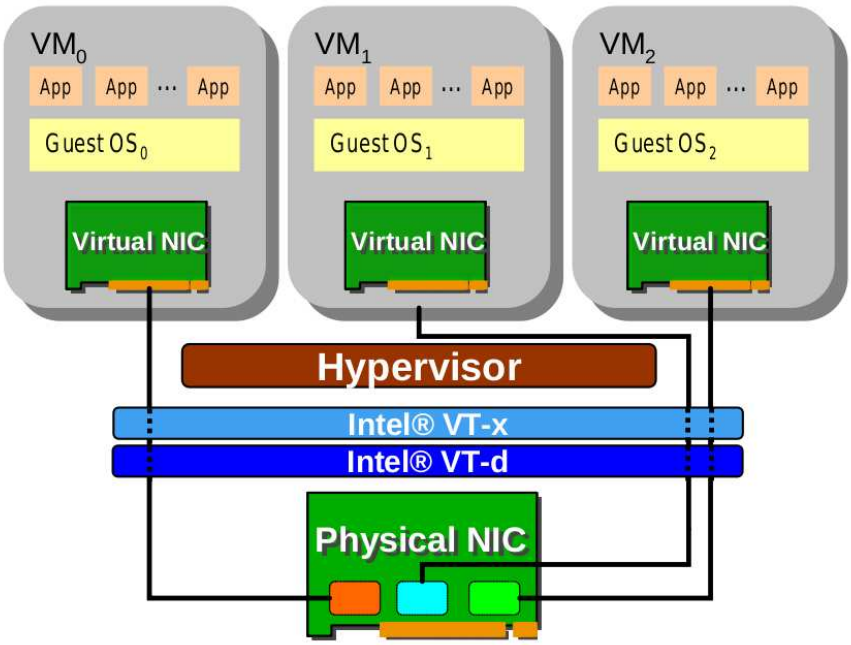

FIG. 4.4 – Architecture de virtualisation *Single Root I/O* [sri].

qui permet à l'unité de traitement de entrées/sorties (comme l'I/O MMU sur Intel ou les PAMU sur le P4080) de distinguer les différents flux d'informations et d'appliquer des translations aux accès mémoires et aux interruptions concernant la PF et les VFs.

L'intérêt de cette technologie est surtout un gain en performance. Effectivement, les requêtes PCIe sont directement acheminées entre les machines virtuelles et les VFs associées, par DMA, sans nécessiter une émulation logicielle par l'hyperviseur (cette émulation induit un surcoût en performance). Cela permet également de limiter la taille de l'hyperviseur ainsi que sa complexité.

Pour pouvoir exploiter pleinement la technologie SR-IOV, il y a un certain nombres de contraintes que l'architecture matérielle, en particulier la MMU et l'I/O MMU, doivent respecter. Tout d'abord, il est nécessaire que l'I/O MMU soit capable de distinguer les opérations réalisées par les différentes cartes virtuelles (associées à des machines virtuelles différentes). Pour cela, il est nécessaire que l'I/O MMU implémente ses contrôles d'accès à la granularité de la fonction PCIe pour que les différentes machines virtuelles n'aient accès qu'à leur carte virtuelle dédiée (caractérisée par leur VF). Ceci permet d'autoriser très précisément les accès DMA notamment, en fonction de l'identifiant de VF inclus dans la requête PCIe. Il est également nécessaire que la gestion des interruptions par l'I/O MMU soit effectuée finement : elle doit être capable de distinguer les interruptions provenant des cartes virtuelles (c'est-à-dire des différentes VFs) de façon à les rediriger correctement vers la machine virtuelle associée (on parle d'*IRQ remapping*). Ensuite,

l'utilisation correcte de SR-IOV impose aussi des contraintes sur la gestion des ressources au niveau de la MMU. En effet, la configuration des différentes VFs est projetée en mémoire en utilisant la technologie MMIO (*Memory Mapped IO*). Or, il est fondamental que chaque machine virtuelle puisse accéder à la configuration de sa VF mais surtout qu'elle ne puisse pas accéder à la configuration des VFs des autres machines virtuelles. Aussi, dans la mesure où la granularité des protections de la mémoire gérées par la MMU est la page (de taille 4ko), il est fondamental que les différentes VFs puissent être mappées sur des pages mémoires différentes de façon à bénéficier des protections dédiées par la MMU.

Ces exigences, ainsi que quelques autres, que nous ne citons pas ici pour ne pas entrer dans trop de détails d'implémentations, sont bien détaillées dans un article de 2013 de Münch *et al.* ([MIM+13]).

4.3.3 Utilisation de SR-IOV dans un contexte avionique embarqué

Dans un contexte avionique utilisant le partitionnement, l'adoption d'un périphérique de type SR-IOV doit s'adapter à l'architecture logicielle utilisée. Ainsi, dans ce contexte, chaque machine virtuelle correspond à une partition différente. Ces différentes partitions non privilégiées, appelées partitions "enfants", ont accès chacune à une carte virtuelle différente représentée par sa VF. L'association de ces VFs aux partitions est réalisée par un logiciel privilégié, typiquement le noyau du système embarqué lui même, ou une partition de contrôle privilégiée, ainsi que le propose par exemple Münch *et al.* dans [MIM+13]. Cette partition de contrôle est associée à la fonction physique PF et peut ainsi avoir accès à l'ensemble des ressources de la carte ainsi qu'à sa configuration.

Le partitionnement spatial est réalisé en utilisant la MMU et l'I/O MMU. Dans une architecture SR-IOV, nous avons listé ci-dessus un certain nombre de contraintes techniques qui doivent être respectés pour que leur utilisation soit efficace. Ces contraintes permettent en fait de séparer les contextes des partitions, car une partition ne doit pas être capable d'avoir accès au contexte (ensemble de registres) d'une autre partition. De son côté, l'hyperviseur ou la partition de contrôle doit gérer ses propres variables mais aussi des variables globales pour les autres partitions.

Si les contraintes listées ci-dessus concernant la MMU et l'I/O MMU sont respectées, l'isolation spatiale est assurée efficacement entre partitions. Si l'on prend l'exemple de la plateforme d'expérimentation utilisée dans cette thèse, basée sur le P4080, ces contraintes ne sont pas respectées puisqu'elle inclut bien une MMU et un certain nombre de PAMUs mais ces PAMUs ne sont pas capables d'établir des contrôles d'accès basés sur des identifiants de fonctions PCIe. Les PAMUs sont capables de faire des vérifications uniquement en fonction de l'identité PCIe du périphérique lui-même. Ainsi, dans le cadre de l'utilisation du P4080, il est nécessaire d'enrichir le noyau du système embarqué de façon à pallier ce problème.

De même, en ce qui concerne les interruptions, il est nécessaire, comme nous l'avons invoqué précédemment, que l'architectrure offre la possibilité de gérer l'*IRQ Remapping* de façon fine, basée sur l'identification de fonctions PCIe, ce qui encore une fois n'est pas le cas de la plateforme basé sur le P4080 utilisée dans cette thèse et qui implique

donc une prise en charge supplémentaire du noyau de système embarqué.

Enfin, le partitionnement temporel doit s'effectuer avec la prise en compte des durées d'exécution de la carte réseau elle-même pour traiter les paquets. Cette durée d'exécution doit avoir un impact minimal sur la durée d'exécution des partitions. En effet, en l'absence de périphérique SR-IOV, habituellement les entrées/sorties sont gérées par une partition I/O qui doit donc traiter les problèmes d'accès concurrents des différentes partitions au périphérique, et qui doit donc assurer l'isolation spatiale et temporelle. Dans le cas d'un périphérique SR-IOV, les différentes partitions accèdent directement à leur périphérique virtuel et c'est donc au périphérique de gérer l'isolation temporelle directement. Il est donc fondamental d'éviter qu'une partition ayant un accès légitime à sa carte virtuelle (via la VF associée) puisse empêcher une autre partition de pouvoir émettre ou recevoir dans son quantum de temps imparti, de façon à assurer le partitionnement temporel. Cela nécessite en particulier que le périphérique offre la possibilité au noyau du système embarqué (ou à la partition de contrôle en fonction du choix d'architecture effectué) d'annuler ou de suspendre une commande (opération d'envoi ou de réception demandée à la carte) mais aussi de gérer l'accès simultané de plusieurs partitions.

En conclusion de cette section, on peut affirmer que si l'ensemble des contraintes relatives à la MMU et l'I/O MMU sont bien respectés sur la plateforme matérielle, l'utilisation d'un composant de type SR-IOV, en particulier pour les communications AFDX des systèmes embarqués avioniques, s'avère être une solution efficace (à la fois en termes de performance et en termes de complexité du logiciel privilégié) et sécurisée (car matériellement isolée en espace et en temps). Nous estimons donc que son utilisation dans les systèmes avioniques du futur est à considérer avec soin.

Conclusion

En conclusion, nous avons proposé dans ce chapitre différents moyens pour se protéger de tentatives de corruptions réalisées par une partition malveillante s'exécutant sur un calculateur avionique. Nous avons présenté des contre-mesures spécifiques pour les attaques expérimentées dans le chapitre 3. Nous avons ensuite proposé, pour certaines de ces attaques, des contre-mesures génériques applicables à différents types de systèmes embarqués. Nous avons également justifié l'intérêt de complémenter systématiquement ces contre-mesures par des techniques d'analyse statique de code et par des techniques d'analyse dynamique. Enfin, nous avons proposé une architecture sécurisée de péripérique partagé entre différentes partitions de criticités différentes. Cette architecture est basée sur les technologies matérielles de virtualisation *Single Root-I/O*. Nous avons présenté son utilisation dans un contexte avionique embarqué. L'ensemble de ces contre-mesures nous paraît être une bonne base pour améliorer la conception et le développement sécurisés de systèmes embarqués avioniques du futur.

Conclusion générale

La sécurité des systèmes informatiques est une préoccupation grandissante depuis déjà de nombreuses années. De nouveaux programmes malveillants sont sans cesse découverts, ces programmes étant par ailleurs de plus en plus complexes et de plus en plus difficiles à éradiquer. En même temps, les systèmes informatiques sont de plus en plus connectés à de multiples réseaux, filaires ou non, qui constituent autant de "portes d'entrées" possibles pour les attaquants. Si la sécurité des systèmes informatiques "de bureau" fait l'objet de nombreux travaux depuis de nombreuses années, il n'en est pas de même pour les systèmes embarqués. Ces systèmes, embarqués dans tous nos appareils connectés à Internet à la maison par exemple, mais aussi embarqués dans les voitures, dans les avions, sont eux aussi et de plus en plus aujourd'hui, la cible potentielle d'attaques. Les travaux que nous avons menés dans le cadre de cette thèse visent à apporter une contribution à l'amélioration de la sécurité des systèmes embarqués, en particulier des systèmes avioniques. En effet, l'évolution récente de ces systèmes, notamment avec l'adoption par les avionieurs de l'architecture IMA *Integrated Modular Avionics*, vise à réduire les coûts de développement et de maintenance, mais elle accroît potentiellement la surface d'attaques possible.

Les travaux présentés dans ce manuscrit ont donc eu pour but d'analyser et d'améliorer les mécanismes de sécurité existants sur des systèmes embarqués avioniques. Même si la sécurité de ces systèmes est déjà prise en compte par les avioneurs, les mécanismes de sécurité utilisés aujourd'hui sont davantage basés sur des méthodes d'analyse statique de code et sur l'utilisation de méthodes formelles pour la conception des applications avioniques. Ces méthodes sont évidemment essentielles et très pertinentes mais elles nécessitent, selon nous, d'être complétées par des analyses de vulnérabilités, en particulier, visant les couches basses du logiciel. C'est dans cet objectif que se sont situés les travaux de cette thèse.

Nous avons proposé dans le premier chapitre de ce manuscrit, après avoir justifié l'intérêt des nos travaux, une méthodologie pour mener à bien cette analyse de vulnérabilités. Elle a été réalisée en fonction d'une classification des attaques sur les systèmes embarqués, et c'est une première contribution de cette thèse. Cette classification, présentée en détails dans le second chapitre, comprend deux catégories : les attaques ciblant les fonctionnalités de base et les attaques ciblant les mécanismes de tolérance aux fautes. Les attaques ciblant les fonctionnalités de base sont similaires aux attaques visant les systèmes informatiques grand public et concernent les attaques ciblant le processeur, la

gestion de la mémoire, les communications, la gestion du temps, la gestion et l'ordonnancement des processus, les mécanismes cryptographiques et les fonctions ancillaires. Les attaques ciblant les mécanismes de tolérance aux fautes mettent l'accent sur des attaques qui visent à exploiter les mécanismes habituels de tolérance aux fautes présents dans la majorité des systèmes embarqués critiques et en particulier dans les systèmes avioniques. Ces attaques ciblent particulièrement les mécanismes que sont le traitement d'erreurs et le traitement de fautes.

Dans le troisième chapitre, nous avons mené des expérimentations concernant un certain nombre des attaques identifiées dans la classification du second chapitre, sur un système embarqué expérimental développé par Airbus. Ce système nous a été fourni avec ses sources, ce qui nous a permis notamment de réaliser une analyse de code afin d'identifier d'eventuelles vulnérabilités. Nous avons considéré un système avionique composé de différentes partitions de criticités différentes et nous avons fait l'hypothèse qu'une partition non critique peut être malveillante et tenter de corrompre une partition critique s'exécutant sur le même calculateur (comme il est permis dans le contexte de l'architecture IMA). Ces attaques ont été réalisées sur une plateforme QorIQ P4080 de Freescale. Nous avons réalisé des attaques ciblant la gestion de la mémoire, la gestion du temps, les communications et les mécanismes de tolérance aux fautes. Ces attaques ont donné des résultats intéressants pour la plupart et ont permis de corriger certaines vulnérabilités présentes dans le système, ce qui a contribué à améliorer la sécurité du système. Ce cercle vertueux nous a paru fondamental car il a été réalisé durant la phase de développement. Il constitue pour nous la seconde contribution de cette thèse.

Enfin, nous avons proposé, dans le quatrième chapitre de ce manuscrit, des contre-mesures spécifiques à nos attaques réalisées et des contre-mesures génériques adaptables à d'autres types de systèmes embarqués. Nous avons également donné des recommandations générales pour les développeurs d'applications embarquées et proposé une architecture sécurisée de périphérique partagé entre applications de différentes criticités.

Bilan et perspectives

Nous voudrions, avant d'aborder les perspectives, établir un petit retour d'expérience concernant les expérimentations et la collaboration avec Airbus dans le cadre de ces travaux. Nous avons eu la grande chance de disposer d'un exécutif temps-réel expérimental avec ses sources, développé par Airbus. Les échanges que nous avons effectués ont été très fructueux et ont permis d'introduire des modifications de cet exécutif pendant la phase de développement, en vue d'améliorer sa sécurité. De ce point de vue, nous estimons que cette étude a été particulièrement intéressante et fructueuse. En revanche, il nous a été assez difficile de nous procurer la plateforme d'expérimentation comprenant le P4080, les sondes JTAG et CodeWarrior, ce qui nous a un peu limité dans la quantité des expérimentations que nous avons pu réaliser ensuite. La découverte de vulnérabilités dans le noyau a été un exercice très intéressant, même si, le noyau étant particulièrement minimaliste et statique (Airbus le qualifie de "nano-noyau"), il était probable qu'il serait difficile de trouver des vulnérabilités très sérieuses. Il ne contient en effet que les fonctions

vitales, pour ordonnancer les partitions et gérer les interruptions, et toute sa configuration est statique. La surface d'attaque était donc très limitée depuis une partition utilisateur. Cependant, il a été intéressant de comprendre comment le noyau temps-réel et les partitions ont été développés ainsi que le fonctionnement du P4080 de Freescale et ses nombreuses caractéristiques. Nous avons pu réaliser quelques expérimentations avec succès sur les communications, la gestion du temps et les mécanismes de tolérance aux fautes, et nous avons pu éliminer ces vulnérabilités avec la collaboration d'Airbus, ce qui était un des objectifs de cette thèse. L'important est que ces expérimentations et cette plateforme nous ont permis de valider, dans un contexte spécifique, notre méthodologie de recherche de vulnérabilités.

En ce qui concerne les perspectives à ces travaux, on peut envisager des perspectives de deux ordres. Les premières perspectives, à court terme et sur un plan technique, concernent les expérimentations que nous avons menées. En effet, faute de temps et faute de matériel adéquat, nous n'avons pu toutes les réaliser. A titre d'exemple, d'autres attaques ciblant la gestion de la mémoire pourraient être mises en œuvre, notamment visant les caches du processeur. Ces attaques, que nous n'avons pu mener par faute de temps nous semblent particulièrement pertinentes dans un contexte avionique. D'autres attaques peuvent être envisagées si l'on dispose du matériel adéquat pour réellement mettre à l'épreuve certains mécanismes de tolérance aux fautes, en particulier basés sur la redondance. Nous ne disposions pas, pour cette étude, de multiples exemplaires du même calculateur et nous n'avions donc pas la possibilité de mettre en œuvre certaines techniques de tolérance aux fautes basées sur la redondance.

A plus long terme et sur un plan méthodologique, nous pouvons envisager des perspectives concernant la méthodologie d'analyse de vulnérabilités pour des systèmes embarqués avioniques que nous avons présenté dans le chapitre 1.6. Il serait intéressant d'étudier l'adéquation de cette méthodologie à d'autres domaines d'applications. Le domaine automobile par exemple en est un bon exemple car de nombreux systèmes embarqués sont présents dans les voitures d'aujourd'hui. De même que pour les avions, la connectivité des véhicules automobiles devient de plus en plus étendue (connectivité Wifi, BlueTooth, GSM, etc.) et toutes ces sources de connectivité sont autant de sources d'attaques possibles. L'adoption de COTS dans ces véhicules peut également introduire les mêmes types de risques que leur adoption dans les systèmes avioniques. Il est donc pertinent de vouloir améliorer les mécanismes de sécurité existants sur les automobiles. Il est probable que la classification que nous avons proposée doive être adaptée aux mécanismes de tolérance aux fautes des véhicules, qui sont différentes des mécanismes des systèmes avioniques. On peut bien sûr envisager ce même genre d'études sur d'autres systèmes embarqués très différentes, comme les systèmes embarqués qui envahissent notre quotidien professionnel et personnel aujourd'hui, car il est probable que beaucoup de ces équipements n'ont pas été développé avec un réel souci de sécurité.

Enfin, une des perspectives de cette thèse concerne les contre-mesures. En particulier, nous avons proposé une première réflexion concernant une architecture sécurisée de périphérique partagé entre différentes partitions. Cette architecture, utilisant des mécanismes matériels incontournables, est selon nous, une contribution intéressante à la

conception d'architectures sécurisées de systèmes avioniques critiques. Cette réflexion demande à être poursuivie, en ayant toujours le souci d'envisager des mécanismes de sécurité à la fois logiciels et matériels pour pouvoir concevoir des mécanismes de défense efficaces contre des attaques visant les couches les plus basses du logiciel.

Bibliographie

[01N13] 01Net. Une attaque informatique bloque l'Internet chinois pendant des heures. `http://www.01net.com/editorial/601826/une-attaque-informatique-bloque-linternet-chinois-pendant-des-heures/`, août 2013.

[A+06] J. Akoka et al. *Encyclopédie de l'informatique et des systèmes d'information.* Vuibert, décembre 2006.

[Aci06] Kaya Koç C. Aciiçmez, O. Trace-Driven Cache Attack on AES. In *ICICS'06 Proceedings of the 8th international conference on Information and Communications Security*, pages 112–121, 2006.

[Aci07a] O. Aciiçmez. Yet Another MicroArchitectural Attack : Exploiting I-Cache. In *Workshop on Computer Security Architecture (CSAW'07)*, pages 11–18, New-York City (NY, USA), 2007. ACM.

[Aci07b] Schindler W. Kaya Koç C. Aciiçmez, O. Cache Based Remote Timing Attack on the AES. In *CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286. Springer-Verlag, 2007.

[Aed12] J. Aedla. Linux Kernel CVE-2012-0056 Local Privilege Escalation Vulnerability. Technical report, Janvier 2012.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *Transactions On Dependable and Secure Computing*, 1 :11–33, janvier/mars 2004.

[ari05] ARINC 811 : Commercial Aircraft Information Security Concepts of Operation and Process Framework. Technical report, ARINC Corp., décembre 2005.

[ARI08] ARINC 653 : Avionics Application Software Standard Interface Part 2 Extended Services. Technical report, ARINC Corp., décembre 2008.

[ari09] ARINC 664 : Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network. Technical report, ARINC Corp., septembre 2009.

[arp10] ARP 4754A : Guidelines For Development Of Civil Aircraft and Systems. Technical report, SAE International, décembre 2010.

[Art14] C. Arthur. Apple's SSL iPhone vulnerability : how did it happen, and what next ? Technical report, Février 2014.

[ASK07] O. Aciiçmez, J.-P. Seifert, and C. K. Koç. Predicting Secret Keys via Branch Prediction. In *The 7th Cryptographers' Track at the RSA Conference on Topics in Cryptlogy (CT-RSA'07)*, pages 225–242, San Francisco (CA, USA), 2007. Springer-Verlag.

[Avi85] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *Transactions On Software Engineering*, 11 :1491–1501, décembre 1985.

[B+02] P. Baudin et al. Caveat : a tool for software validation. In *Dependable Systems and Networks (DSN)*, page 537, Juin 2002.

[B+12a] P. Bieber et al. Security and Safety Assurance for Aerospace Embedded Systems. 2012.

[B+12b] J.-P. Blanquart et al. Similarities and Dissimilarities between Safety Levels and Security Levels. 2012.

[Ber05] D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[Boi06] A. Boileau. Hit by a Bus : Physical Access Attacks with FireWire. In *RUXCON*, Melbourne (Australia), septembre 2006.

[Bon06] Mironov I. Bonneau, J. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 201–215, 2006.

[Bro13] M. Brodbeck. Potential EMI from Portable Electronic Devices (PED) on aircraft. `http://www.serec.ethz.ch/EVENTS%20WEF%202011/AVIATION+SPACE_28JUN13/8_PED%20EMC%20in%20Aviation_V3_BRODBECK.pdf`, juin 2013.

[C+07] P. Cousot et al. Varieties of Static Analyzers : A Comparison with ASTREE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering(TASE'07)*, pages 3–20, Shangai, juin 2007.

[Can12] Radio Canada. Fermeture de Megaupload : Anonymous riposte par une série d'attaques. `http://www.radio-canada.ca/nouvelles/International/2012/01/19/013-anonymous-attaque-sites-megaupload.shtml`, janvier 2012.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symposium Principles Of Programming Languages (POPL'77)*, pages 238–252, Los Angeles (California, Etats-Unis), 1977.

[CC13] X. Chen and D. Caselden. MS Windows Local Privilege Escalation Zero-Day in The Wild. Technical report, Novembre 2013.

[CE09] B. Chelf and C. Ebert. Ensuring the Integrity of Embedded Software with Static Code Analysis. volume 26, pages 96–99, mai/juin 2009.

[Cou13] J. Couzins. Attacking the Hypervisor. Technical report, Novembre 2013.

[DADN12] A. Dessiatnikoff, É. Alata, Y. Deswarte, and V. Nicomette. Potential Attacks on Onboard Aerospace Systems. In *IEEE Security and Privacy vol. 10 N. 4*, Juillet/Août 2012.

[DADN13] A. Dessiatnikoff, É. Alata, Y. Deswarte, and V. Nicomette. Low-level Attacks on Avionics Embedded Systems. In *The 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, Toulouse, France, 24-27 septembre 2013.

[DDAN13] A. Dessiatnikoff, Y. Deswarte, É. Alata, and V. Nicomette. Securing Integrated Modular Avionics Computers. In *The 32nd Digital Avionics Systems Conference (DASC)*, Syracuse (NY, USA), 5-10 octobre 2013.

[DKCG99] Y. Deswarte, M. Kaâniche, P. Corneillie, and J. Goodson. SQUALE Dependability Assessment Criteria. pages 27–38. Springer, 1999.

[DLM09] L. Duflot, O. Levillain, and B. Morin. Getting into the SMRAM : SMM Reloaded. In *CanSecWest 2009*, Vancouver (BC, Canada), 2009.

[DO11] J. D. O'Grady, ZDNet. FAA approves iPads in the cockpit ; American Airlines to start Friday. `http://www.zdnet.com/blog/apple/faa-approves-ipads-in-the-cockpit-american-airlines-to-start-friday/11865`, décembre 2011.

[do310] DO-326 Airworthiness Security Process Specification. Technical report, RTCA Inc., décembre 2010.

[Dor04] M. Dornseif. 0wned by an iPod - hacking by Firewire. In *PacSec/core04*, Tokyo (Japan), novembre 2004.

[Duf07] L. Duflot. *Contribution à la sécurité des systèmes d'exploitation et des microprocesseurs.* , Thèse de doctorat de l'université de Paris XI, octobre 2007.

[ESZ08] S. Embleton, S. Sparks, and C. Zhou. SMM Rootkits : A New Breed of OS Independant Malware. In *4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*. ACM, 2008.

[Fre11] Freescale. P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. Technical report, avril 2011.

[Gen13] S. Genevès. *Améliorations de performances des systèmes multi-coeur : environnement d'exécution événementiel efficace et étude comparative de modèles de programmation.* , Thèse de doctorat de l'université de Grenoble, avril 2013.

[GST13] D. Genkin, A. Shamir, and E. Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. Technical report, décembre 2013.

[HS09] N. Henninger and H. Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *29th Annual International Cryptology Conference (Crypto 2009*, volume 5677, pages 1–17, Santa Barbara (CA, USA), 2009. Springer-Verlag.

[IW08] V. Igure and R. Williams. Taxonomies of Attacks and vulnerabilities in Computer Systems. volume 10, pages 6–19, 1st quarter 2008.

[JH08] E. Jaeger and T. Hardin. A few remarks about formal development of secure systems. *In : HASE, IEEE Computer Society*, pages 165–174, 2008.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. volume 1666, pages 388–397. Springer, 1999.

[Koc96] P. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS,and Other Systems. Technical report, 1996.

[KW07] R. Kaiser and S. Wagner. The PikeOS Concept - History and Design. , 2007.

[L+96] J.-C. Laprie et al. *Guide de la sûreté de fonctionnement, 2ème édition.* Cépaduès-éditions, 1996.

[LA95] S. Liu and R. Adams. Limitations of Formal Methods and An Approach to Improvement. In *Software Engineering Conference*, pages 498–507, Brisbane, Qld., Australie, décembre 1995.

[Laa09] Y. Laarouchi. *Sécurités (immunité et innocuité) des architectures ouvertes à niveaux de criticité multiples : application en avionique.* , Thèse de doctorat de l'université de Toulouse, novembre 2009.

[Lab12] Kaspersky Lab. Pourquoi la complexité est le pire ennemi de la sécurité informatique. Technical report, Kaspersky, 2012.

[LABK90] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures. *IEEE Computer*, 23 :39–51, juillet 1990.

[LAEJ11] M. Lee, M. Ahn, and K. Eun Jung. Fast Secure Communications in Shared Memory Multiprocessor Systems. volume 22, pages 1714–1721, juillet 2011.

[LD11] Nicomette V. Lacombe, E. and Y. Deswarte. Enforcing kernel constraints by hardware-assisted virtualization. volume 7, pages 1–21, Février 2011.

[lem13] AFP. Il sera bientôt possible de laisser son téléphone allumé dans l'avion. `http://www.lemonde.fr/technologies/article/2013/12/09/il-sera-bientot-possible-de-laisser-son-telephone-allume-dans-l-avion_3527966_651865.html`, décembre 2013.

[LSLND10] F. Lone-Sang, E. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU Vulnerability. In *5th International Conference on Malicious and Unwanted Software (MalWare 2010)*, pages 9–16, Nancy (France), 2010.

[M+10] J. McCune et al. TrustVisor : Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[May05] D. Maynor. 0wn3d by everything else - USB/PCMCIA Issues. In *CanSecWest/core05*, Vancouver (Canada), mai 2005.

[MIM+13] D. Münch, O. Isfort, K. Muller, M. Paulitsch, and A. Herkersdorf. Hardware-Based I/O Virtualization for Mixed Criticality Real-Time Systems Using PCIe SR-IOV. pages 706–713, Sydney (NSW), décembre 2013.

[Mor91] M. J. Morgan. Integrated Modular Avionics for next generation commercial airplanes. volume 1, pages 43–49, mai 1991.

[Mun11] C. Munoz. Computer Virus Infects Predator Ground Stations. http://breakingdefense.com/2011/10/computer-virus-infects-predator-ground-stations/, octobre 2011.

[NEV01] http://ec.europa.eu/research/growth/aeronautics-days/pdf/posters/nevada.pdf, 2001.

[OST06] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures : the Case of AES. In *The 6th Cryptographers' Track at the RSA Conference on Topics in Cryptlogy (CT-RSA'06)*, pages 1–20, San Jose (CA, USA), 2006. Springer-Verlag.

[OWA13] OWASP. OWASP WebScarab Project. https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project, 2013.

[PAM01] http://www.airbus.com/innovation/eco-efficiency/aircraft-end-of-life/pamela/, 2001.

[Per05] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[pip09] PIPAME, La chaîne de valeur dans l'industrie aéronautique. Technical report, septembre 2009.

[PM88] S. K. Park and K. W. Miller. Random number generators : good ones are hard to find. volume 31, pages 1192–1201. ACM, octobre 1988.

[Pua11] I. Puaut. Worst-Case Execution Time (WCET) estimation : from monocore to multi-core architectures. http://www.ensta-paristech.fr/~chapoutot/seminaire/supports/isabelle_puaut_110517.pdf, mai 2011.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1 :220–232, juin 1975.

[Rus99] J. Rushby. Partitioning in Avionics Architectures : Requirements, Mechanisms, and Assurance. , 1999.

[RW09] J. Rutkowska and R. Wojtczuk. Attacking SMM Memory via Intel® CPU Cache Poisoning. Technical report, mars 2009.

[S+09a] T. Shinagawa et al. BitVisor : A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)*, pages 121–130. ACM, 2009.

[S+09b] J. Souyris et al. Formal Verification of Avionics Software Products. In A. Cavalcanti and D. Dams (Eds), editors, *FM '09 Proceedings of the 2nd World Congress on Formal Methods*, pages 532–546, Eindhoven, Pays-Bas, novembre 2009. Springer-Verlag Berlin Heidelberg.

[Sai05] A. Saidane. *Conception et réalisation d'une architecture tolérant les intrusions pour les serveurs Internet.* , Thèse de doctorat de l'université de Toulouse, janvier 2005.

[SCA11] `http://www.scarlettproject.eu/partnership/default.asp`, 2011.

[Ska13] P. Skaves. Information for Cyber Security Issues Related to Aircraft Systems rev-A. In *32nd Digital Avionics Systems Conference*, octobre 2013.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development of Embedded Systems. In *In Advances in Object-Oriented Information Systems, Lecture*, pages 298–312. Springer-Verlag, 2002.

[SPL95] O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 processor architecture : pitfalls for secure systems. In *IEEE Symposium on Security and Privacy*, pages 211–222, Oakland (CA, USA), 1995.

[sri] SR-IOV Architecture. `http://msdn.microsoft.com/en-us/library/windows/hardware/hh440238%28v=vs.85%29.aspx`.

[Tes13] H. Teso. Aircraft Hacking : Practical Aero Series. In *Hack In The Box*, avril 2013.

[usc12] Government Accountability Office. Cybersecurity : Threats Impacting the Nation. `http://www.gao.gov/assets/600/590367.pdf`, avril 2012.

[VIC01] `http://www1.fsr.tu-darmstadt.de/research/projects/en_victoria.html`, 2001.

[vir14] Multiples vulnérabilités dans Oracle VirtualBox et Apache Tomcat. Technical report, Février 2014.

[vmw14] Multiples vulnérabilités dans les produits VMware. Technical report, janvier 2014.

[WFLH04] X. Wang, D. Feng, X. Lai, and Yu H. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. 2004.

[xen14] Multiples vulnérabilités dans Xen. Technical report, Février 2014.

[You] Y. Younan. 25 years of Vulnerabilities : 1988-2012. Technical report.

[ZBR07] K. Zhou, B. Breinhauer, and T. Rausch. Violations of Real Time Communication Constraints caused by Memory Transfers exceeding CPU Cache Limits in RTAI and Rtnet. In *5th International Conference on Industrial Informatics*, volume 1, pages 267–272, Vienne (Autriche), juin 2007. IEEE.

[ZY09] W. Zhang and J. Yan. Accurately Estimating Worst-Case Execution Time for Multi-Core Processors with Shared Direct-Mapped Instruction Caches. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 455–463, Beijing, Chine, août 2009.

Résumé

L'évolution actuelle des systèmes embarqués à bord des systèmes complexes (avions, satellites, navires, automobiles, etc.) les rend de plus en plus vulnérables à des attaques, en raison de : (1) la complexité croissante des applications ; (2) l'ouverture des systèmes vers des réseaux et systèmes qui ne sont pas totalement contrôlés ; (3) l'utilisation de composants sur étagère qui ne sont pas développés selon les méthodes exigées pour les systèmes embarqués critiques ; (4) le partage de ressources informatiques entre applications, qui va de pair avec l'accroissement de puissance des processeurs. Pour faire face aux risques de malveillances ciblant les systèmes embarqués, il est nécessaire d'appliquer ou d'adapter les méthodes et techniques de sécurité qui ont fait leurs preuves dans d'autres contextes : Méthodes formelles de spécification, développement et vérification ; Mécanismes et outils de sécurité (pare-feux, VPNs, etc.) ; Analyse de vulnérabilités et contre-mesures. C'est sur ce dernier point que portent nos travaux de thèse.

En effet, cet aspect de la sécurité a peu fait l'objet de recherche, contrairement aux méthodes formelles. Cependant, il n'existe pas actuellement de modèle formel capable de couvrir à la fois des niveaux d'abstraction suffisamment élevés pour permettre d'exprimer les propriétés de sécurité désirées, et les détails d'implémentation où se situent la plupart des vulnérabilités susceptibles d'être exploitées par des attaquants : fonctions des noyaux d'OS dédiées à la protection des espaces d'adressage, à la gestion des interruptions et au changement de contextes, etc. ; implémentation matérielle des mécanismes de protection et d'autres fonctions ancillaires. C'est sur ces vulnérabilités de bas niveau que se focalise notre étude.

Nos contributions sont résumées par la suite. Nous avons proposé une classification des attaques possibles sur un système temps-réel. En nous basant sur cette classification, nous avons effectué une analyse de vulnérabilité sur un système réaliste : une plateforme avionique expérimentale fournie par Airbus. Il s'agit d'un noyau temps-réel critique ordonnancé avec plusieurs autres applications, le tout exécuté sur une plateforme Freescale QorIQ P4080. C'est à travers une application dite « malveillante », présente parmi l'ensemble des applications, que nous essayons de modifier le comportement des autres applications ou du système global pour détecter des vulnérabilités. Cette méthode d'analyse de vulnérabilités a permis de détecter plusieurs problèmes concernant les accès mémoire, la communication entre applications, la gestion du temps et la gestion des erreurs qui pouvaient conduire à la défaillance du système global. Enfin, nous avons proposé des contre-mesures spécifiques à certaines attaques et des contre-mesures génériques pour le noyau temps-réel qui permet d'empêcher une application d'obtenir des accès privilégiés ou encore de perturber le comportement du système.

Mots-clés: systèmes embarqués, sécurité informatique, contre-mesure, système avionique, analyse de vulnérabilité.

Abstract

Security is becoming a major concern for embedded computing systems in various critical industrial sectors (aerospace, satellites, automotive, etc.). Indeed, recent trends in the development and operation of such systems, have made them more and more vulnerable to potential attacks, for the following reasons : 1) increasing complexity of the applications ; 2) openness to applications and networks that are note completely under control ; 3) Use Commercial-Off-The-Shelf (COTS) hardware and software components ; 4) Resource sharing among different applications, driven by the increase of processors capabilities.

To improve the security of such systems, it is necessary to apply or adapt methods and techniques that have proven their efficiency in other contexts : Formal methods for specification, development and verification ; Security mechanisms and tools (firewalls, VPNs, etc.) ; Vulnerability assessment and countermeasure provision.

The research carried out in this thesis adresses the latter technique. This aspect of security analysis cannot be easily covered by current formal methods, since no existing model is able to cover both high-level abstractions, where security properties can be defined, and low-level implementation details, where most vulnerabilities that could be exploited by attackers lie : OS kernel implementation of address space protection, interrupt management, context switching, etc. ; hardware implementation of protection mechanisms and other ancillary functions. Very few research projects are addressing this aspect of security, which is the main objective of this thesis. In particular, our research focuses on low-level vulnerabilities, but contrarily with common practice, we aim to discover and analyze them during the development process.

Our contributions are summarized as follows. We elaborated a classification of low-level vulnerabilities for different implementations of real-time embedded systems. Based on this classification, we carried out a vulnerability analysis study on a realistic system : An experimental avionic platform provided by Airbus. It consists of a critical real-time kernel scheduling the execution of different applications on a freescale QorIQ P4080 platform. The identification and analysis of vulnerabilities is carried out based on a "malicious" application hosted on the platform that attempts to corrupt the behavior of the other applications or the global system considering different types of low level attacks. Such experiments allowed us to identify some problems related to the management of memory accesses, the communication between applications, time management and error handling that could lead to the global system failure.We have also proposed generic counter measures to protect the real-time kernel against specific attacks, and to prevent a given application from escalating its privileges or trying to compromise the system behavior.

Keywords: embedded systems, computer security, countermeasure, avionic system, vulnerability assessment.