Sécurité par analyse comportementale de fonctions embarquées sur plateformes avioniques modulaires intégrées

Aliénor Damien

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :

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

Présentée et soutenue le *11/06/2020* **par :**

ALIÉNOR DAMIEN

Sécurité par analyse comportementale de fonctions embarquées sur plateformes avioniques modulaires intégrées

---

JURY

MOHAMED KAÂNICHE	Directeur de recherche	Président du Jury
VALÉRIE VIET TRIEM TONG	Professeur des Universités	Rapporteur
OLIVIER FESTOR	Professeur des Universités	Rapporteur
ASSIA TRIA	Responsable scientifique	Examinateur
MICHAËL HAUSPIE	Maître de conférences	Examinateur
VINCENT NICOMETTE	Professeur des Universités	Directeur de thèse
ERIC ALATA	Maître de conférences	Co-directeur de thèse
NATHALIE FEYT	Ingénieur de recherche	Co-directeur de thèse

École doctorale et spécialité :

MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :

Laboratoire d'analyse et d'architecture des systèmes

Directeur(s) de Thèse :

Vincent NICOMETTE, *Éric ALATA* et *Nathalie FEYT*

Rapporteurs :

Valérie VIET TRIEM TONG et *Olivier FESTOR*

Remerciements

Au cours de ces quelques années de thèse, j'ai eu l'occasion de rencontrer et de côtoyer de nombreuses personnes qui ont chacune contribué, à leur manière, à la réussite de ces travaux. Je tiens à les remercier ici pour tout ce qu'elles m'ont apporté, humainement, techniquement, scientifiquement, ...

J'adresse tout d'abord mes remerciements à Mohamed KAÂNICHE, pour m'avoir accueillie au sein de l'équipe Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF) du Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS), pour avoir suivi mes travaux avec attention tout au long de ce parcours et m'avoir appris la rigueur que demande le travail de recherche. Je suis très honorée qu'il ait également accepté de présider mon jury le jour J.

Je remercie très chaleureusement mes directeurs de thèse, officiels ou non, qui ont participé à l'aboutissement de ces travaux. Tout d'abord, je remercie mon directeur de thèse, Vincent NICOMETTE, pour son soutien indéfectible, sa bonne humeur, ses mots bien choisis au moment où j'en ai eu besoin. Au-delà de l'aspect scientifique, il m'a toujours permis de dépasser mes doutes et d'avancer malgré les difficultés et je le remercie énormément pour cela. Je remercie également Eric ALATA, mon co-directeur de thèse, pour m'avoir apporté des réponses pragmatiques à mes interrogations, pour ses idées constructives et pour nos discussions animées. Je remercie également Nathalie FEYT, Michaël MARCOURT et Marc FUMEY pour m'avoir accordé leur confiance et m'avoir donné les moyens de mener à bien mes travaux de recherche. J'ai particulièrement apprécié être challengée régulièrement, tant sur le plan technique que sur l'aspect communication.

Mes remerciements vont également à Valérie VIET TRIEM TONG et à Olivier FESTOR, qui ont accepté de rapporter ma thèse et de participer à mon jury, et dont les retours m'ont à la fois confortée sur tout le travail effectué pendant cette thèse mais également permis d'ouvrir de nouvelles perspectives pour la suite. Je remercie également Assia TRIA et Michaël HAUSPIE pour avoir accepté de participer à mon jury de thèse, pour vos questions et retours pertinents et chaleureux après avoir "dévoré" mon manuscrit. En tant que femme, je suis particulièrement honorée de citer sur ma couverture de thèse trois femmes aussi reconnues dans le domaine de la sécurité que vous, Assia, Nathalie, Valérie.

Je tiens également à remercier les équipes de Thales avec qui j'ai eu l'occasion d'interagir, qui m'ont tout appris sur ce domaine passionnant qu'est l'avionique. Je pense en particulier à Stéphane M., Marc G., Julien B., Frédérik B., Nicolas L., Christian S., Olivier B., qui ont suivi mes travaux, ont pris le temps de m'expliquer ce qu'ils savaient sur les architectures avioniques, ont relus mes papiers, ou encore ont alimenté mes pistes de réflexion. Je pense également à Frédérique, Sophie, Arnaud, Nicolas, Fabien, Vincent, Quentin, Géraud, Christophe, Sandrine, Anne, Dominique, Daniel, Erwan, Pascal, Gilles, Joël, Didier, Tarik, Philippe, Pierre-Noël, Joël, ... J'oublie certainement de citer certains d'entre vous, mais sachez que je vous suis très reconnaissante pour tout ce que vous m'avez apporté. Je remercie

également très chaleureusement Théo CUSNIR et Yacine SMINI qui m'ont aidée à réaliser les différentes expérimentations au cours de leurs stages et qui m'ont permis d'avancer concrètement à un moment décisif.

Je remercie également toute l'équipe TSF pour m'avoir accueillie avec bienveillance, pour ses conseils avisés, pour m'avoir appris à développer mon sens critique, et avoir grandement contribué à équilibrer cette expérience intense. Je pense en particulier à Lola, Thierry, Clément et Pierre-François avec qui j'ai eu le plaisir de partager mon bureau, je vous remercie beaucoup pour ces discussions animées sur des sujets plus ou moins sérieux, pour avoir supporté mes sautes d'humeur, et surtout pour votre soutien dans les moments difficiles. Mes remerciements vont également à tous mes compagnons de galère qui sont passés par cette équipe ou qui y sont encore, Carla, Matthieu, Guillaume, Jonathan, Rémi, Daniel, Kalou, Christophe, Yuxiao, Benoît, Rui, Florent, Alexandre, Romain, Malcolm, Cyrius, Jean, Bilel, Mohamed, Luca, Raul. Je suis heureuse d'avoir pu tisser des liens forts avec vous tous, et de pouvoir vous considérer pas seulement comme collègues, mais surtout comme amis.

Je remercie également Sarah, Clara, Nadège, Camille, et Filou pour m'avoir sortie régulièrement, et m'avoir rappelé qu'il existe une vie hors de la thèse !

Mes derniers remerciements vont à ma famille et particulièrement à mes parents et à mes deux frères qui m'ont toujours soutenue dans cette aventure, qui ont fait des efforts pour essayer de comprendre ce que je faisais et qui m'ont toujours aidée à prendre du recul. Aymeric, Tristan, vous avez toujours été ma meilleure source d'inspiration. Papa, Maman, ma plus grande motivation a toujours été de vous rendre fiers.

Pour finir, je tiens à remercier celui qui m'a donnée envie de me lancer dans l'aventure, mon fiancé, William. Sans toi, je n'aurais jamais été au bout.

À ma grand-mère, qui aurait sûrement versé une larme pour sa petite flûtiste. Tu me manques.

Table des matières

Table des figures

Liste des tableaux

Liste des Abréviations

ACD	Aircraft Control Domain, page 17
AISD	Airline Information Services Domain, page 17
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information, page 30
API	Application Programming Interface, page 11
ARINC	Aeronautical Radio INCorporated, page 9
ARP	Aerospace Recommended Practice, page 9
ATM	Air Traffic Management, page 29
BITE	Built-In Test Equipment, page 68
CIFRE	Conventions Industrielles de Formation par la REcherche, page 2
COTS	Commercial Off-The-Shelf, page 10
DAL	Design Assurance Level, page 20
DGAC	Direction Générale de l'Aviation Civile, page 8
DO	DOcument, page 9
EASA	European Union Aviation Safety Agency, page 8
ED	Eurocae Documents, page 9
EUROCAE	European Organisation for Civiel Aviation Equipment, page 9
FAA	Federal Aviation Administration, page 8
FC	Failure Condition, page 29
FHA	Functional Hazard Assessment, page 29
FLS	Field Loadable Software, page 12
FMES	Failure Mode Effect Summary, page 29
FN	False Negative, page 37
FP	False Positive, page 37
HIDS	Host-based Intrusion Detection System, page 31
HMM	Hidden Markov Model, page 36
IA	Intelligence Artificielle, page 32
IDS	Intrusion Detection System, page 31

IFE	In-Flyt Entertainement, page 16
IHM-DA	Interface Homme-Machine pour les Données de l'Aéronef, page 70
IHM-DV	Interface Homme-Machine pour les Données de Vol, page 70
IMA	Integrated Modular Avionics, page 10
IP	Internet Protocol, page 29
LDACS	L-band Digital Aeronautical Communications System, page 29
MAF	MAjor Frame, page 11
MMU	Memory Management Unit, page 11
NIDS	Network-based Intrusion Detection System, page 31
OCSVM	One-Class Support Vector Machine, page 36
PCA	Principal Components Analysis, page 133
PIESD	Passenger Information and Entertainment Services Domain, page 17
PKI	*Public Key Infrastructure*, page 18
POD	Passenger-Owned Devices, page 17
RTCA	Radio Technical Commission for Aeronautics, page 9
RTOS	Real-Time Operating System (Système d'exploitation temps-réel), page 12
SAE	Society of Automotive Engineers, page 9
SDA	Security Domain of the Application, page 44
SOC	Security Operations Center, page 38
SSA	System Safety Assessment, page 29
SVM	Support Vector Machine, page 51
TADS	Technical Assistance Database System, page 16
TN	True Negative, page 37
TP	True Positive, page 37
VL	Virtual Link, page 11
WCET	Worst Case Execution Time, page 65

Introduction

L'aviation est l'un des modes de transport les plus sûrs (au sens de l'anglais *safety*). La sûreté du vol est la priorité des acteurs du domaine. Historiquement, les systèmes avioniques critiques, conçus pour atteindre un niveau de sûreté élevé, étaient de fait considérés comme non concernés par les failles de sécurité logicielles au sens *security*. En effet, de tels systèmes étaient physiquement isolés du monde extérieur, avec des capacités de connectivité très limitées et en s'appuyant sur des protocoles et logiciels propriétaires. Par conséquent, la probabilité qu'un attaquant puisse accéder à ces systèmes pour y exploiter une faille était considérée comme nulle. Des attaques récentes sur des systèmes embarqués ont amené à remettre en question ces certitudes. Les systèmes embarqués modernes disposent désormais de multiples points d'entrée qui peuvent se révéler vulnérables à de potentielles attaques. La conjonction de l'introduction de nouvelles technologies, souvent COTS (*Commercial Off-The-Shelf*), de l'interconnexion des différents systèmes, et de l'évolution des menaces informatiques présente donc un risque pour la sécurité des systèmes avioniques.

Le piratage informatique d'un avion ou hélicoptère en vol représente une réelle menace pour la sécurité du transport aérien. « Le piratage informatique d'un avion est possible », « La cybercriminalité représente bien une véritable menace pour le transport aérien » a affirmé le directeur de l'Agence Européenne de Sécurité Aérienne (AESA), Patrick Ky [Les Echos 2015]. Le directeur exécutif de l'AESA a confirmé les craintes que l'on pouvait nourrir dans ce domaine. **Les menaces sont multiples.** Les attaquants peuvent être motivés par l'intention de nuire, le vol de renseignements, le profit, ou encore, dans une certaine mesure, la promotion d'objectifs politiques (motivations "hacktivistes"). **Les adversaires sont nombreux.** Ils attaquent en permanence sur de multiples fronts, ont des objectifs multiples, et travaillent dans l'anonymat. En plus d'adversaires externes, pour lesquels des solutions existent déjà et sont en permanente amélioration, nous devons également protéger les systèmes embarqués contre les attaques internes, qui peuvent provenir d'actes intentionnels malveillants ou d'utilisations non prévues des systèmes.

Dans ce contexte, la compréhension des risques liés à des attaques malveillantes et l'identification des solutions permettant de mettre en évidence les vulnérabilités potentielles et de détecter les tentatives d'intrusion dans les systèmes avioniques sont cruciales pour maintenir un haut niveau de sûreté. Afin d'améliorer la sécurité de ces systèmes, il est possible d'utiliser des mécanismes de **protection**, qu'ils soient techniques (cryptographie, pare-feux, VPN, etc) ou organisationnels (gestion et formation des utilisateurs, gestion des accès, etc). Ces mécanismes permettent de protéger un environnement considéré comme "de confiance" vis à vis d'attaques venant de l'extérieur. Il est également possible d'utiliser des mécanismes de **détection**, qui vont s'attacher à détecter une attaque informatique, en vue de lever une alerte et/ou de bloquer cette attaque durablement. La détection nécessite 1)

des moyens de surveillance des systèmes avec des outils de supervision spécifiques au domaine concerné, 2) des mécanismes de qualification, consistant à confirmer l'attaque et à la comprendre (quelles actions on pu être menées ? que est l'impact de l'attaque ?) et 3) des mécanismes de correction, qui visent à apporter les contre-mesures nécessaires.

La combinaison entre l'exploitation par un attaquant d'une vulnérabilité résiduelle non identifiée à ce jour et le contournement des barrières de protection est une éventualité qui doit être considérée. A l'image de ce qui se fait dans les systèmes d'information classiques, les recherches de pointe dans le domaine des logiciels antivirus, des firewalls de nouvelle génération, ou des systèmes de prévention d'intrusion, sont tout à fait nécessaires, mais parfois insuffisantes à maintenir l'assiégeant hors du périmètre à protéger. Il est ainsi indispensable d'identifier des parades aux menaces non prévues, qu'elles soient issues de l'extérieur ou de l'intérieur.

Dans ce cadre, l'analyse comportementale semble être un axe de recherche prometteur pour détecter des intrusions ciblant les applications critiques temps-réel. En effet, les propriétés temporelles déterministes des applications temps-réel embarquées peuvent être un véritable atout quant à la détection d'un changement de comportement. Une attaque sur de tels équipements modifierait vraisemblablement l'exécution temps-réel des applications étant donné qu'une application malveillante présenterait un changement de comportement vis-à-vis de son environnement (système d'exploitation, autres applications). En particulier, un changement de comportement relatif à sa consommation des ressources systèmes (cycles CPU, mémoire, ...) devrait être observé. De plus, l'analyse comportementale permet de se focaliser sur des comportements applicatifs connus (sains), et donc de prévoir la détection de nouvelles attaques sans nécessiter de connaissances préalables sur des exemples d'attaques réelles.

Cette thèse, menée dans le cadre d'un contrat CIFRE (Conventions Industrielles de Formation par la REcherche) entre Thales AVS et le LAAS-CNRS, présente nos réflexions autour de l'applicabilité des techniques d'analyse comportementale pour la détection d'intrusion sur des applications avioniques temps-réel. Ces travaux s'articulent autour de quatre thèmes :

1. l'intégration d'un système de détection d'intrusion comportemental dans le processus actuel de développement d'une application avionique,
2. les outils nécessaires au paramétrage d'une telle solution,
3. les contraintes d'implémentation à prendre en compte pour une solution embarquable, et
4. les données à relever en cas d'alerte permettant de faciliter la qualification des alertes.

Le premier chapitre présente des éléments de contexte indispensables à la bonne compréhension des spécificités d'un environnement avionique, et notamment les différences fondamentales qui empêchent l'application directe de solutions existantes pour des systèmes d'information plus classiques comme celui d'une entreprise.

Le second chapitre propose un aperçu des travaux existants dans le domaine de la sécurité pour l'avionique, et plus précisément les travaux relatifs aux systèmes de détection d'intrusion pour les systèmes embarqués, ainsi que les travaux relatifs aux quatre axes explorés dans cette thèse.

Les chapitres suivants présentent les travaux relatifs à chacun des thèmes évoqués précédemment. Le troisième chapitre détaille notre approche, permettant d'intégrer une solution de détection d'intrusion comportementale au sein d'un calculateur avionique, en se basant sur les phases d'intégration et d'opération du processus actuel de développement d'une application avionique dans une architecture IMA (*Integrated Modular Avionics*). La description de cette approche est complétée par la proposition d'un outil d'injection de code malveillant permettant de tester l'efficacité d'une solution de détection d'intrusion dans un environnement avionique. A notre connaissance, il n'existe pas de base de données d'applications avioniques malveillantes disponible publiquement, et il est important d'avoir des exemples d'attaques pour calibrer au mieux notre système de détection d'intrusion comportemental.

Le chapitre suivant détaille une première implémentation de l'approche et de l'outil d'injection, afin de valider la mise en oeuvre de notre approche sur un cas d'étude précis. Ce chapitre décrit les différents composants nécessaires à la mise en place de notre système de détection d'intrusion comportemental, et notamment le processus permettant de définir le modèle de comportement normal d'une application avionique. Ce processus implique la mise en oeuvre de l'outil d'injection, dont l'implémentation est également détaillée. Enfin, ce chapitre se conclut par deux expérimentations permettant de montrer la pertinence de notre système de détection d'intrusion ainsi que de notre outil d'injection de code malveillant.

Le cinquième chapitre va plus loin dans la définition de l'approche en étudiant différentes alternatives pour le choix des modèles d'apprentissage utilisés pour modéliser le comportement normal de l'application. Ces différentes alternatives sont envisagées vis à vis de leur intérêt du point de vue de l'embarquabilité (facilité à appliquer aux aéronefs actuels, consommation CPU faible, besoin en mémoire faible, interprétabilité des résultats). Une première évaluation de ces alternatives est effectuée avec une implémentation "hors ligne" (le minimum est effectué sur le calculateur cible afin de réduire les coûts de test), permettant une première caractérisation en terme de précision des résultats, ainsi que quelques ordres de grandeur quant à leur consommation de ressources. Deux solutions sont ensuite sélectionnées pour être développées entièrement sur le calculateur cible, afin de connaître précisément leur consommation en terme de ressources.

Enfin, le dernier chapitre explore les différentes informations pouvant être utiles pour l'analyse d'une alerte, et propose un premier niveau de classification de l'alerte en vol en fonction des anomalies relevées, ainsi qu'une hiérarchisation des informations à conserver en fonction des ressources de stockage disponibles. Ces informations seront ensuite conservées et traitées au sol pour une investigation plus approfondie de l'alerte.

Chapitre 1

Contexte et concepts fondamentaux

Sommaire

Le domaine avionique présente plusieurs particularités, que nous proposons de présenter succinctement dans ce chapitre. Les concepts nécessaires à la compréhension des prochains chapitres y sont détaillés. Il est important de noter que ce sont toutes ces spécificités qui rendent difficile l'application de technologies déjà éprouvées dans d'autres domaines, au domaine avionique.

Le chapitre commence par rappeler quelques définitions relatives à la sûreté de fonctionnement et plus particulièrement à la sécurité-immunité, et présente les principaux organismes de standardisation et autorités de certification du domaine avionique. Ensuite, une partie dédiée aux architectures des systèmes avioniques permet de comprendre le fonctionnement d'un système avionique et le modèle de menace considéré dans cette thèse. La partie suivante expose les problématiques liées à la sécurité dans ce contexte, en donnant d'une part les dernières évolutions susceptibles d'impacter les systèmes avioniques, et d'autre part les difficultés liées aux systèmes embarqués critiques temps-réel. Ce chapitre se termine par quelques hypothèses supplémentaires considérées dans ces travaux, permettant de donner au lecteur un aperçu complet des objectifs de cette thèse.

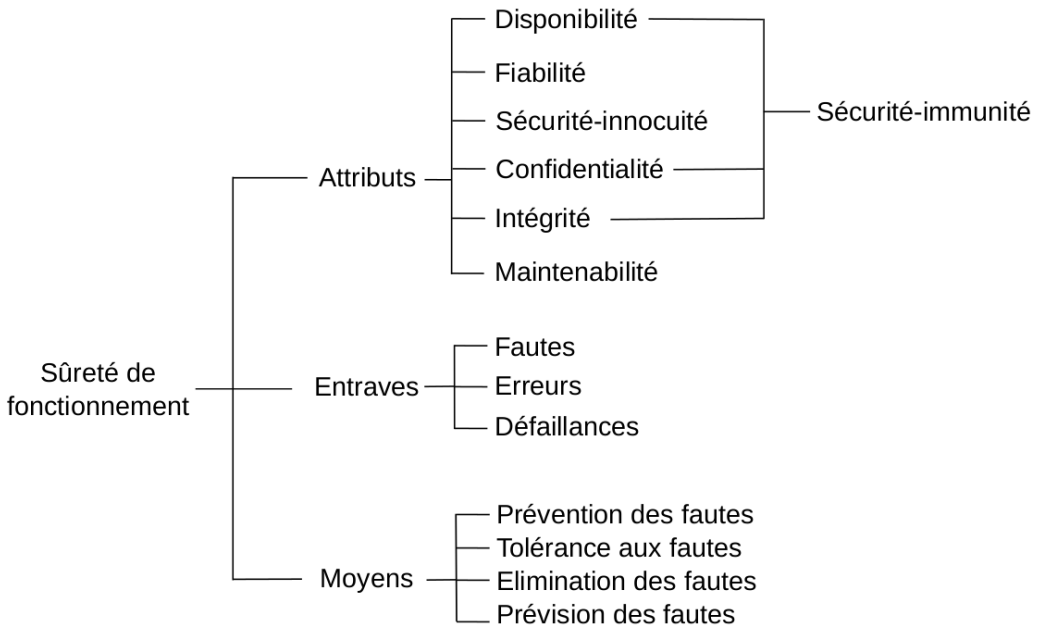


FIGURE 1.1 – L'arbre de la sûreté de fonctionnement

1.1 Définitions

La sûreté de fonctionnement définit les termes de sécurité-innocuité (*safety*) et de sécurité-immunité (*security*). Mes travaux se placent dans le cadre de la sécurité-immunité (*security*). Cette partie donne le vocabulaire relatif à la sûreté de fonctionnement, puis plus précisément celui relatif à la sécurité-immunité. Enfin, les principaux standards et autorités du domaine avionique sont présentés.

1.1.1 La sûreté de fonctionnement

[Laprie 1996] définit la **sûreté de fonctionnement** d'un système informatique comme étant la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il délivre. Le service délivré par un système est son comportement tel qu'il est perçu par son (ou ses) utilisateur(s), l'utilisateur étant un autre système (humain ou physique) qui interagit avec le système considéré. La sûreté de fonctionnement, représentée en arbre sur la Figure 1.1, est décrite selon trois notions : les attributs (définition d'un comportement sûr), les entraves (circonstances indésirables correspondant aux causes ou résultats de la non sûreté de fonctionnement) et les moyens (moyens contribuant à assurer un fonctionnement sûr en dépit des entraves).

Les **attributs** représentent les différentes propriétés selon lesquelles la sûreté de fonctionnement peut être perçue :

— **Disponibilité** : aptitude du système à être prêt à l'utilisation,

— **Fiabilité** : continuité du service,

— **Sécurité-innocuité** : absence de conséquences catastrophiques pour l'environnement,

— **Confidentialité** : absence de divulgations non autorisées de l'information,

— **Intégrité** : absence d'altérations inappropriées de l'information,

— **Maintenabilité** : aptitude aux réparations et aux évolutions.

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. On distingue trois types d'entraves : les défaillances, les erreurs et les fautes. Leur relation peut être illustrée de la façon suivante :

... → Défaillance → Faute → Erreur → Défaillance → Faute → ...

Un **défaillance** (ou **défaillance du système**) survient lorsqu'un service délivré dévie de l'accomplissement de sa fonction. Une **erreur** est la partie d'un état d'un système qui est susceptible d'entraîner une défaillance. Par propagation, plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne. La cause adjugée ou supposée d'une erreur est une **faute**.

Les sources des fautes sont très diverses et peuvent être classées selon différents points de vue :

— la phase de création (fautes de développement ou fautes opérationnelles)

— la situation par rapport aux frontières du système (internes ou externes)

— la cause phénoménologique (naturelle ou humaine)

— l'objectif (faute malicieuse ou non)

— la dimension (faute matérielle ou logicielle)

— l'intention (faute délibérée ou non)

— la capacité (faute accidentelle ou non)

— la persistance (faute permanente ou temporaire)

Enfin, les **moyens** permettant d'assurer un fonctionnement sûr en dépit des entraves sont définis selon quatre catégories :

— **Prévention des fautes** : empêcher l'occurrence ou l'introduction d'une faute

— **Tolérance aux fautes** : fournir un service qui remplit la fonction du système en dépit de fautes

— **Elimination des fautes** : réduire le nombre et la sévérité des fautes

— **Prévision des fautes** : estimer la présence, la création et les conséquences des fautes

1.1.2 La sécurité-immunité

Les attributs de la sûreté de fonctionnement permettent de différencier les termes de *safety* et de *security*. La *safety* (sécurité-innocuité) 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). La *security* (sécurité-immunité) définit les propriétés selon lesquelles un système est dit "sécurisé" (protégé contre les fautes intentionnelles, dites **malveillances**). La *security* est liée aux trois attributs **disponibilité**, **confidentialité**, et **intégrité**, et permet de prendre en compte les menaces informatiques tels que les virus, vers, bombes logiques, etc. Dans la suite de ce manuscrit, les termes *sécurité*, *sécurité informatique* ou *security* feront référence à la sécurité-immunité.

Le projet MAFTIA [Powell 2001] précise le concept de faute due à l'homme pour la sécurité-immunité et décrit deux catégories de fautes associées : les fautes logiques malignes et les intrusions. Les **fautes logiques malignes** sont des fautes intentionnelles conçues pour provoquer des dégâts (bombes logiques) ou pour faciliter les futures intrusions par l'introduction de vulnérabilités. Les fautes logiques malignes peuvent être présentes dès la première utilisation du système ou durant son exploitation par l'installation d'un cheval de Troie ou par une intrusion. Les **intrusions** sont définies conjointement à deux causes :

— Une **attaque** est une faute d'interaction externe au système, dont le but est de violer un ou plusieurs des attributs de sécurité. Elle peut aussi être définie comme une tentative d'intrusion.

— Une **vulnérabilité** est une faute qui peut être accidentelle, intentionnelle malveillante ou non malveillante placée dans les exigences, la spécification, la conception ou la configuration du système, ou dans la manière dont il est utilisé.

— Une **intrusion** est déclenchée par l'exploitation d'une vulnérabilité au cours d'une attaque. C'est donc une faute malveillante, initiée depuis l'extérieur pendant l'utilisation du système.

1.1.3 L'écosystème avionique

Le transport aérien a une forte culture *safety*. En effet, les conséquences d'une défaillance en vol peuvent avoir des conséquences dramatiques sur la vie des passagers. Aujourd'hui, lorsqu'un constructeur souhaite lancer un nouveau programme d'aéronef sur le marché, celui-ci doit satisfaire les exigences d'un processus de certification. Ceci implique de se conformer à des règles de conception strictes, selon un processus de certification surveillé puis validé par une autorité nationale de l'aviation civile. En France, c'est la Direction Générale de l'Aviation Civile (DGAC) qui délivre cette autorisation. En 2002, une autorité européenne a vu le jour sous le nom d'EASA (European Union Aviation Safety Agency), afin d'éditer des certificats de navigabilité pour l'ensemble du territoire de l'union européenne. Son équivalent américain est la FAA (Federal Aviation Administration).

A partir des règles de navigabilité énoncées par les autorités de certification, plusieurs normes et recommandations ont vu le jour afin d'aider les constructeurs à respecter ces règles. Parmi les organismes responsables de la rédaction de ces normes et recommandations, on retrouve notamment : SAE [1], RTCA [2] pour les Etats-Unis et EUROCAE [3] pour l'Europe. Ils fournissent respectivement des documents nommés ARP (Aerospace Recommended Practice), DO (DOcument), et ED (Eurocae Documents). Aujourd'hui, les organismes EUROCAE et RTCA travaillent ensemble afin d'harmoniser leurs recommandations, c'est pourquoi la plupart des documents cités dans ce manuscrit feront référence conjointement à des documents ED et DO. Ces différentes recommandations (ARP, ED, DO) sont devenues des normes avioniques au fil des années.

L'entreprise ARINC (Aeronautical Radio INCorporated), créée en 1929 et détenue jusqu'en 2013 par les principales compagnies aériennes et divers constructeurs aéronautiques américains, propose également les principaux standards de communications à l'intérieur des aéronefs et entre les aéronefs et le sol.

1.2 L'avionique modulaire intégrée (IMA)

Le terme "système avionique" désigne les éléments électroniques et informatiques embarqués à bord d'un aéronef. Ces éléments assurent différentes fonctionnalités parmi lesquelles on peut par exemple citer le traitement des informations provenant des capteurs, le pilotage automatique, la gestion du carburant, les communications bord/sol, ou le contrôle des moteurs. Ces équipements représentent une part importante des coûts de développement d'un aéronef, de l'ordre de 30 à 35% pour un avion commercial [PIPAME 2009].

1.2.1 D'une architecture fédérée à une architecture IMA

Historiquement, les systèmes avioniques étaient construits selon une architecture dite fédérée (cf Figure 1.2), c'est-à-dire qu'une fonction était hébergée sur un calculateur dédié (i.e., un matériel spécifique). Cette architecture permet une gestion simple et efficace des équipements pour répondre aux exigences de sûreté du milieu avionique. Elle facilite notamment la redondance des équipements et minimise les dépendances entre fonctions, ce qui limite d'une part les risques de propagation d'erreurs, et qui permet d'autre part d'isoler les fonctions défaillantes du reste du système [Gatti 2016].

Cependant, ce type d'architecture implique une forte dépendance entre le matériel et le logiciel, ce qui induit par exemple des systèmes de communication très dépendants des applications mais aussi du matériel, un parc de matériel très hétérogène au sein d'un même aéronef, et d'importants coûts de mise à jour, aussi bien logicielle que matérielle.

1. https ://www.sae.org
2. https ://www.rtca.org
3. https ://www.eurocae.net

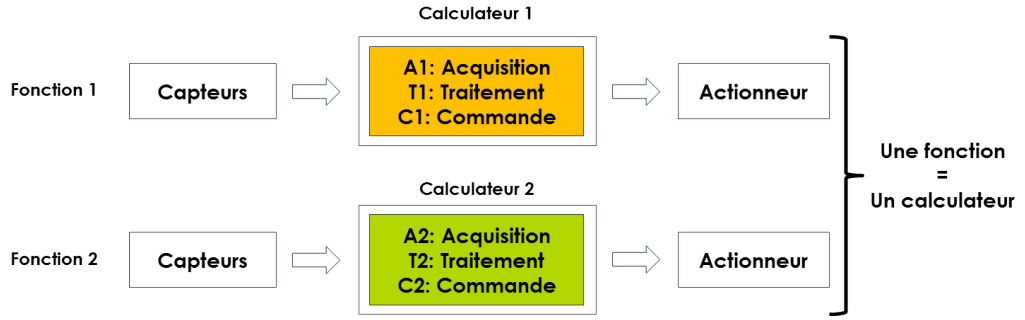


FIGURE 1.2 – Principe de l'architecture fédérée

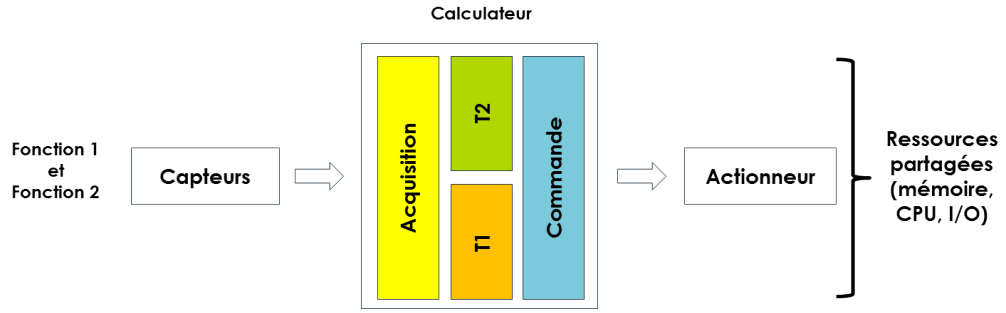


FIGURE 1.3 – Principe de l'architecture IMA

L'architecture IMA propose une nouvelle vision des systèmes avioniques en introduisant une ségrégation entre les composants logiciels et le matériel, ce qui permet de mutualiser les ressources matérielles de l'avion pour plusieurs fonctions avioniques. Cette nouvelle architecture a été mise en œuvre sur les programmes Rafale, Boeing 777, Falcon 7X, A380, A400M, Boeing 787, et plus récemment sur l'A350. La Figure 1.3 illustre cette nouvelle architecture. Les composants communs d'acquisition et de commande des fonctions 1 et 2 ont été mutualisés dans un même calculateur, qui effectue les traitements T1 et T2. Cette nouvelle architecture apporte plusieurs avantages, notamment la réduction du poids de l'avion (moins de composants matériels et moins de câbles), des coûts de maintenance réduits de par l'utilisation de calculateurs génériques, et des coûts de développement réduits par l'utilisation de systèmes standardisés et de COTS (*Commercial Off-The-Shelf*).

1.2.2 Principes fondamentaux de l'IMA

L'un des défis majeurs de l'IMA est sa capacité à garantir les mêmes propriétés de sûreté que pour une architecture fédérée. Pour ce faire, les calculateurs IMA doivent garantir un partitionnement spatial et temporel robuste entre les applications, c'est-à-dire qu'une application qui s'exécute sur un calculateur ne doit en aucun cas impacter une autre application indépendante de la première s'exécutant sur le même calculateur (ou le même réseau de calculateurs). Dans le cas des calculateurs IMA, cette propriété est vérifiée par deux principes décrits dans la norme

FIGURE 1.4 – Allocation de la ressource CPU sur un calculateur avionique

ED-124/DO-297 (*Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*) : la ségrégation spatiale et la ségrégation temporelle.

Ces deux types de ségrégations sont réalisées à l'aide d'une configuration statique de ressources. Notons qu'une application (au sens fonction avionique) peut être composée de plusieurs partitions (au sens logiciel), auxquelles des ressources sont allouées de façon indépendante.

La ségrégation spatiale consiste à séparer physiquement les ressources allouées à chaque partition. Plus précisément, l'espace mémoire et les ports d'entrée/sortie (I/O) sont alloués statiquement à chaque partition. Si un calculateur possède 30 connecteurs I/O, il peut par exemple en réserver 10 à la partition P1, 5 à la partition P2, et 15 à la partition P3. S'il possède 100Mo de mémoire RAM, chaque partition se verra allouer un espace RAM qui est garanti à l'exécution par une unité de gestion mémoire ou MMU (*Memory Management Unit*).

La ségrégation temporelle consiste à partager une même ressource physique sur une durée prédéfinie. C'est le cas pour l'utilisation de la CPU, les calculateurs avioniques actuels étant des calculateurs mono-coeur. L'allocation temporelle de la ressource CPU est illustrée sur la Figure 1.4. Une période de temps commune à l'ensemble des partitions d'un même calculateur, appelée MAF (MAjor Frame), permet d'orchestrer l'exécution des partitions sur une période donnée. Cette orchestration est ensuite répétée en boucle pendant toute la durée d'exécution du calculateur. Au sein de cette période (50ms dans l'exemple), chaque partition se voit allouer une durée et un séquencement. Sur la Figure 1.4, les partitions P1, P2 et P3 se voient allouer respectivement 10ms, 20ms, et 5ms toutes les 50ms.

Concernant l'utilisation du réseau ARINC 664 part 7 ou A664p7 (*Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*), une latence maximale est garantie par l'allocation de *Virtual Link* (ou VL) pour les partitions d'une part, et par la configuration des switchs formant le réseau d'autre part.

L'ARINC 653 (*Avionics Application Standard Software Interface*) spécifie la partie logicielle permettant le partitionnement temporel et spatial pour les systèmes d'exploitation temps-réel avioniques critiques. Elle définit notamment une interface applicative ou API (*Application Programming Interface*) composée de plusieurs catégories de services : la gestion des partitions, la gestion des processus, la gestion du temps, les communication inter-partitions, les communications intra-partitions, et la gestion des erreurs.

1.2.3 Acteurs impliqués

Plusieurs acteurs sont impliqués dans le cycle de vie d'une application, parmi lesquels les fournisseurs d'applications, le fournisseur de calculateurs, l'intégrateur calculateur, l'intégrateur système, l'avionneur, ou encore les compagnies aériennes.

1.2.3.1 Rôles des acteurs

Le fournisseur de calculateur développe un calculateur conforme à l'architecture IMA, doté d'un système d'exploitation temps-réel (RTOS) certifié (COTS ou propriétaire) et des outils associés. En particulier, il doit proposer les services décrits par l'ARINC 653. Le développement de la plateforme matérielle et du RTOS peuvent éventuellement être effectués par deux acteurs différents. Les fournisseurs d'applications sont en charge du développement logiciel des fonctions avioniques. L'intégrateur calculateur alloue les ressources des calculateurs disponibles aux différentes applications, suit leur développement, et vérifie l'intégration technique des applications sur les calculateurs. L'intégrateur calculateur ne vérifie donc pas les fonctionnalités des applications, mais seulement l'utilisation correcte des ressources qui leur ont été allouées. L'intégrateur système et l'avionneur sont en charge de l'intégration fonctionnelle inter-applications. Une fois l'aéronef en opération, c'est à la compagnie aérienne de prévoir et de réaliser les éventuelles mises à jour. La Table 1.1 résume ces différents rôles, ainsi que les phases de vie de l'aéronef pendant lesquelles ils interviennent.

1.2.3.2 Interactions entre acteurs

Les interactions entre ces acteurs sont décrites Figure 1.5.

Dans un premier temps, le fournisseur de calculateur développe un calculateur intégrant un RTOS répondant aux spécifications de l'ARINC 653 [Prisaznuk 2008]. Il édite un document appelé **Domaine d'Usage** qui décrit le cadre d'utilisation pour lequel le calculateur a été certifié. Ce document explicite aussi bien des règles de codage (ex : vérification des types, identifiants uniques, ...) que des règles destinées à l'intégrateur calculateur (ex : nombre maximal de partitions supporté) ou à l'intégrateur système (ex : capacité réseau disponible). Ces règles sont vérifiées à différents niveaux, via des outils automatiques ou manuellement.

Une fois le calculateur disponible et les ressources globales de l'avion connues, l'intégrateur calculateur répartit ces ressources entre les différentes applications commandées par l'avionneur. Il fournit un *Contrat d'insertion* [Conmy 2003] à chaque fournisseur d'application, qui décrit formellement les ressources pouvant être utilisées par l'application (mémoire allouée, I/O alloués, partitions allouées). Chaque fournisseur développe ensuite son application selon cette allocation, et retourne à l'intégrateur calculateur un fichier téléchargeable. Ce fichier téléchargeable, appelé FLS (*Field Loadable Software*) est un ensemble de code compilé contenant à la fois le code applicatif de la partition avionique correspondante, mais aussi ses données de configuration. L'intégrateur calculateur récupère ces différents FLS et

TABLE 1.1 – Rôles des acteurs impliqués dans le cycle de vie d'un système avionique

Acteur	Phase	Rôle
Fournisseur de calculateur	1 Développement du calculateur	Développe la plateforme matérielle, le RTOS, et les outils associés
Fournisseurs d'application	2 Développement des applications	Développent la partie logicielle des fonctions avioniques demandées par l'avionneur
Intégrateur calculateur	2 Développement des applications	Alloue les ressources matérielles disponibles aux différentes applications
	3 Intégration	Installe les applications sur les calculateurs, vérifie la bonne allocation des ressources, et édite un rapport prouvant les propriétés de ségrégation spatiale et temporelle de l'ensemble (calculateur + applications)
Intégrateur système et avionneur	3 Intégration	Intégration fonctionnelle de l'ensemble de l'aéronef
Compagnie aérienne	4 Opération	Opère et effectue la maintenance de l'aéronef

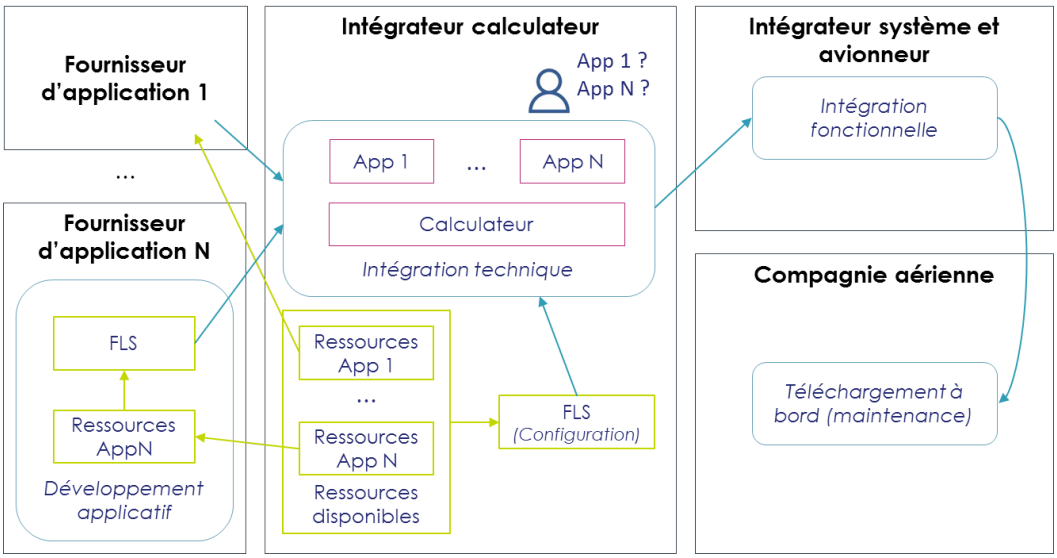


FIGURE 1.5 – Interactions entre les acteurs du développement d'un aéronef selon une architecture IMA

installe l'ensemble sur un calculateur pour vérifier la bonne intégration des applications les unes à côté des autres, et notamment le respect du **Domaine d'usage**. Cette intégration est uniquement technique, c'est-à-dire que l'intégrateur calcu-

lateur vérifie uniquement le bon usage des ressources, pas les fonctionnalités des applications. Ces dernières ont été vérifiées au préalable par les fournisseurs d'application.

Un deuxième niveau de vérification fonctionnelle, plus global, est effectué lors de l'intégration système, puis lors de l'intégration avion. Cette vérification consiste notamment à vérifier le bon fonctionnement de l'ensemble des fonctions de l'avion, les unes avec les autres, par exemple lors d'essais en vol. Enfin, l'aéronef est livré à la compagnie aérienne, qui sera responsable de sa mise à jour tout au long de son cycle de vie. Ces mises à jour sont effectuées par des opérateurs de maintenance qualifiés, partout dans le monde.

1.2.3.3 Risques de sécurité et modèle de menace

Cette organisation, de par le grand nombre d'acteurs impliqués et leurs fortes interactions, présente plusieurs faiblesses du point de vue de la sécurité. En effet, la multiplication des acteurs (à titre d'exemple, un A350 peut compter jusqu'à 35 applications, et plus de 50 compagnies aériennes l'utilisent aujourd'hui [4]) implique une plus grande surface d'attaque. Si la menace d'un attaquant externe peut être potentiellement bloquée aux interfaces de l'avion avant d'atteindre les systèmes avioniques, une attaque interne pourrait avoir de graves conséquences sur la sûreté du vol. C'est le modèle de menace qui est considéré dans cette thèse. Plus particulièrement, on peut décrire deux scénarios impliquant un attaquant interne chez un fournisseur d'application, ou chez une compagnie aérienne :

1. la modification malveillante d'un FLS avant son envoi à l'intégrateur calculateur, et
2. la modification malveillante d'un FLS lors d'une mise à jour.

Dans les deux cas, on peut imaginer aussi bien un opérateur malveillant, l'utilisation d'un dispositif de stockage corrompu, ou l'utilisation d'un dispositif de communication altéré. D'autres scénarios relatifs à des attaques externes pourraient avoir le même type de conséquences, mais ne sont pas directement considérés dans cette thèse. La menace prise en compte ici est donc celle d'un FLS modifié de façon malveillante qui serait chargé sur un calculateur.

Dans une architecture aussi statique qu'une architecture avionique, la modification malveillante d'un applicatif devrait, selon toute vraisemblance, générer des modifications dans l'utilisation des ressources calculateur. Dans ce contexte, le rôle de l'intégrateur calculateur est primordial. Dans un premier temps, il partage la responsabilité d'assurer la ségrégation spatiale et temporelle entre applications avec le fournisseur de calculateur, ce dernier décrivant les règles du **Domaine d'usage** dont le premier doit assurer le bon respect par l'ensemble qu'il intègre. Ensuite, il doit également s'assurer que les applications n'ont pas d'interactions non prévues les unes avec les autres, et il constitue un point central par lequel tous les FLS

4. https ://www.airbus.com/aircraft/market/orders-deliveries.html

transitent lors de la phase d'intégration. Son rôle est central, c'est donc le **point de vue adopté dans cette thèse**.

L'une des difficultés de ce positionnement est la potentielle confidentialité des documents relatifs aux applications. En effet, il est rare que l'intégrateur calculateur possède des connaissances fonctionnelles sur les applications qu'il doit intégrer (spécifications, code source). Généralement, ses connaissances se limitent au FLS de l'application et au contrat d'insertion échangé précédemment. En pratique, la mise en oeuvre de l'approche proposée dans cette thèse pourrait être distribuée entre l'intégrateur calculateur, l'intégrateur système, et/ou l'avionneur.

1.3 Problématiques de sécurité dans l'avionique

Cette section présente dans un premier temps les tendances actuelles concernant l'évolution des menaces sur les systèmes embarqués, l'évolution des systèmes avioniques, et l'évolution de la réglementation aéronautique concernant la sécurité. Ensuite, un aperçu des contraintes spécifiques aux systèmes embarqués critiques temps-réel est donné.

1.3.1 Contexte actuel

Les évolutions technologiques de ces dernières années ont permis plus de connectivité à bord des avions. En 2008, le Dreamliner (Boeing 787) est présenté comme le premier avion commercial doté de capacités de connectivité ou "e-enabled aircraft" [Boeing 2009]. Au-delà de la réduction des coûts motivée par ces avancées, les compagnies aériennes cherchent à améliorer en continu leur expérience utilisateur, ou encore à diminuer leur impact sur l'environnement. Cependant, les menaces informatiques profitent également de cette connectivité accrue, et la sécurité des systèmes d'information avion, sans grand impact jusqu'à présent, est un sujet important à prendre en compte pour le futur.

1.3.1.1 Evolution des menaces

Ces dernières années ont vu émerger de nouvelles menaces informatiques sur des systèmes isolés ou embarqués. Par exemple, le ver *Stuxnet* [Chen 2011] est parvenu à perturber les systèmes de commande hors ligne d'une centrale nucléaire iranienne en 2010. Il a ainsi démontré la possibilité de développer des logiciels malveillants sophistiqués afin d'atteindre un but très précis, visant en particulier des infrastructures critiques. En 2015, la prise de contrôle d'une Jeep Cherokee depuis l'extérieur [Greenberg 2015] a également démontré l'existence de nouveaux vecteurs d'attaque liés à des équipements embarqués de plus en plus connectés.

Concernant le domaine aéronautique, des acteurs se lancent spécifiquement à la recherche de failles de sécurité sur des équipements avioniques afin de faire réagir l'industrie aéronautique. On peut citer par exemple la présentation d'Hugo Teso à la conférence Hack in the Box de 2014 (Amsterdam) [Teso 2013], sur ses travaux

de reconstruction en laboratoire d'un cockpit avionique et prise de contrôle sur cet environnement depuis l'extérieur.

Plus récemment, en 2017, des tests d'intrusion commandités par le gouvernement américain ont été effectués avec succès sur le Boeing 757 [Biesecker 2017]. En deux jours, les équipes de Robert Hickey ont été capables d'identifier et d'exploiter des vulnérabilités pour établir une présence dans le système de l'avion, sans accès physique à celui-ci. En 2019, un test d'intrusion effectué sur le F-15, avec accès physique à l'avion cette fois, a montré plusieurs vulnérabilités critiques pouvant affecter le système TADS (*Technical Assistance Database System*) [Marks 2019]. Ce système recueille des images et d'autres informations à partir des capteurs de l'avion. En 2019 également, la société IOActive a découvert des vulnérabilités dans le firmware d'une passerelle réseau utilisée sur le Boeing 787 [Santamarta 2019]. Ce firmware était exposé sur internet, ce qui a permis à la société de le télécharger et d'y appliquer des techniques de rétro-ingénieurie. Une attaque théorique des systèmes avioniques via l'extérieur, exploitant les vulnérabilités précédemment trouvées, a été présentée lors de la conférence Black Hat USA 2019.

1.3.1.2 Evolution des systèmes avioniques

En paralèlle, les systèmes avioniques évoluent également pour répondre à de nouveaux défis tels que la connectivité des passagers, la diminution de la consommation énergétique, l'optimisation des trajectoires, ou la mise à jour de données extérieures en cours de vol (par exemple, les données météo). Ces défis poussent les systèmes avioniques vers plus de connectivité, que ce soit avec l'extérieur (connexion internet aux passagers, mises à jour "over-the-air" des équipements avion, partage de plan de vol) ou à l'intérieur de l'avion (architecture IMA inter-connectée à travers tout l'avion, partage de données de vol aux passagers via les systèmes de divertissement IFE (*In-Flyt Entertainement*), mutualisation des systèmes de communication avec l'extérieur). On retrouve également plus d'équipements de type COTS, permettant de réduire les durées et les coûts de développement.

Ces évolutions, si elles sont bien maîtrisées d'un point de vue *safety*, tendent cependant à élargir la surface d'attaque d'un point de vue *security*. En effet, plus de connectivité implique plus de portes d'entrée potentielles dans les systèmes, et plus de possibilités de trouver des vulnérabilités à l'intérieur du réseau de l'avion. L'utilisation de COTS implique également une maîtrise moins complète des équipements. Si aucun cas de prise de contrôle malveillante d'un avion en vol n'est à déplorer aujourd'hui, il faut cependant être conscient que la menace est réelle et que ce n'est plus une question de "si" mais de "quand" est-ce que cela va arriver.

1.3.1.3 Evolution de la réglementation

Le transport aérien est aujourd'hui l'un des modes de transport les plus sûrs. Les derniers rapports IATA sur la sûreté des vols [IATA 2018] montrent une diminution significative du nombre d'accidents sur le long terme, avec une année record en 2017

(seulement 19 décès à bord pour plus de 4 milliards de voyageurs sur 41.8 millions de vol). Ces chiffres résultent d'une culture historique de la sécurité-innocuité, ancrée depuis longtemps dans l'ensemble des processus de développement et d'opération des systèmes avioniques. En ce qui concerne la sécurité-immunité, de nombreuses mesures physiques permettent de l'atteindre aujourd'hui, par exemple par des mesures de protection d'accès aux aéroports, des procédures d'embarquement, et la qualification des opérateurs aéronautiques. Le développement de mesures logicielles ou matérielles est plus récent, et se doit de protéger l'avion contre des menaces telles que décrites Section 1.3.1.1, au vu des nouveaux vecteurs d'attaque potentiellement introduits par l'évolution des systèmes telle que décrite Section 1.3.1.2.

L'ARINC 811 (*Commercial Aircraft Information Security Concepts of Operation and Process Framework*), édité en 2005, propose une base de définitions quant à la sécurité des systèmes d'information des aéronefs commerciaux, ainsi que des recommandations pour sa mise en œuvre. Ce document, à l'attention des constructeurs (développement de l'aéronef) et des compagnies aériennes (opération de l'aéronef), n'est pas directement utile pour se protéger contre une attaque informatique ou pour éviter l'exploitation de vulnérabilités présentes dans des applications. Cependant, il définit le cycle de vie de la configuration de l'aéronef, les modes des systèmes, les rôles de sécurité, les objectifs de sécurité de l'information, etc.

En particulier, l'ARINC 664 part 5 ou A664p5 (*Aircraft Data Network, Part 5, Network Domain Characteristics and Interconnection*) définit quatre domaines de réseaux dans un avion.

— **ACD (Aircraft Control Domain)** : C'est celui dans lequel se trouvent les applications de commande et de contrôle de l'appareil, notamment les calculateurs d'une architecture IMA. Ce domaine est le plus critique de l'avion.

— **AISD (Airline Information Services Domain)** : Ce domaine regroupe les différents supports relatifs au vol, à la cabine, et à la maintenance de l'aéronef. Ces services, bien que moins critiques que ceux du domaine ACD, restent très critiques vu leur impact potentiel sur l'exploitation de l'appareil (notamment la partie maintenance).

— **PIESD (Passenger Information and Entertainment Services Domain)** : Les communications avec les passagers sont effectuées dans ce domaine. Il y a notamment les services de gestion des écrans de divertissement (système IFE) et les interfaces de connexion des périphériques des passagers.

— **POD (Passenger-Owned Devices)** : Ce dernier domaine considère l'ensemble des équipements appartenant aux passagers (ordinateurs portables, smartphones, consoles de jeu). Dans les avions les plus récents, il existe même des interfaces spécifiques du domaine PIESD permettant de connecter ces équipements à un réseau de la compagnie aérienne, par exemple pour proposer une connexion internet aux passagers.

L'application de ces deux standards se retrouve par exemple sur l'A380 et l'A350, avec la mise en place de diodes réseau pour séparer les domaines de criticité différente, et la mise en place de protections périphériques comme une PKI (Public Key

Infrastructure) permettant de distribuer les mises à jour des systèmes avioniques de façon sécurisée.

Depuis 2014, trois standards de l'EUROCAE ont été publiés, à partir des recommandations éditées dans l'ARINC 811. Plus précisément, l'ED-202/DO-326 (*Airworthiness Security Process Specification*) adresse les aspects de certification relatifs à la sécurité, l'ED-203/DO-356 (*Airworthiness Security Methods and Considerations*) propose des méthodes et recommandations pour démontrer la sécurité de l'aéronef tout au long de son cycle de vie, et l'ED-204/DO-355 (*Information Security Guidance for Continuing Airworthiness*) présente des recommandations pour la sécurisation du cycle de vie de l'aéronef (opération, support, maintenance, administration et déconstruction).

1.3.2 Contraintes spécifiques à l'embarqué critique temps-réel

Le besoin de mettre en place des mesures de sécurité n'étant plus à prouver, certaines contraintes sont cependant à prendre en compte afin de comprendre les difficultés que peut représenter la mise en place de ces mesures. Les parties suivantes proposent tout d'abord un aperçu des ressources disponibles sur avion et les contraintes amenées par le côté "embarqué". Ensuite, les problématiques de certification seront expliquées en détail avec quelques chiffres pour comprendre les contraintes industrielles associées. Finalement, la caractéristique temps-réel de ces systèmes implique aussi des spécificités quant au développement de mesures de sécurité efficaces.

1.3.2.1 Les limitations des systèmes embarqués

Comme tout système embarqué, les ressources matérielles disponibles à bord d'un aéronef sont limitées. On peut évoquer la notion de coût, par exemple, un espace mémoire robuste aux rayonnements électromagnétiques et aux températures extrêmes étant beaucoup plus cher qu'un espace mémoire qu'on retrouve habituellement sur ordinateur. L'espace disponible et le poids sont également également limitant, ce qui impacte les possibilités en terme d'énergie électrique et de puissance de calcul embarquable. Enfin, un aéronef étant amené à voyager dans le monde, son environnement sera amené à changer. Au-delà des aspects météorologiques, la connectivité ne sera pas la même au-dessus de Paris qu'en survolant l'océan atlantique, par exemple. Également, les moyens de mise à jour actuels nécessitent que l'avion soit au sol, ce qui a un coût non négligeable pour les compagnies aériennes. A titre d'exemple, les mises à jour les plus fréquentes aujourd'hui ont lieu tous les 28 jours et concernent uniquement des base de données.

L'introduction de mesures de sécurité impactera inéluctablement ces aspects. Par exemple, des mesures logicielles auront besoin de temps de calcul et de mémoire dédiée, donc un calculateur et des équipements spécifiques, donc du poids et des coûts supplémentaires. Il est important de dimensionner les ressources nécessaires pour ces mesures dans une limite acceptable. Dans ces travaux, nous considérons

TABLE 1.2 – Quelques caractéristiques techniques d'équipements avioniques actuels

Nom	Fournisseur	CPU	RAM	Poids
CPIOM-H	Thales	1.33GHz	115Mb	4kg
IPC-8303	Rockwell Collins	1.4GHz	512Mb à 2Gb	5.4kg à 6.8kg
FMC 2975A	GE Aviation	0.8GHz	512Mb	10.9kg
FV-4000	Esterline	0.5GHz	512Mb	8 à 9.3kg

qu'un maximum de 5% des ressources totales disponibles peut être alloué à des mesures de sécurité.

A titre d'exemple, le Tableau 1.2 liste les ressources disponibles sur différents équipements avioniques actuels. En comparaison, on trouve aujourd'hui des ordinateurs de bureau ayant un processeur d'une fréquence de 2GHz à 4.5GHz, avec de multiples coeurs, et des capacités mémoire RAM de 2Gb à 128Gb. La bonne gestion des ressources est donc un élément primordial à prendre en compte pour développer une fonction embarquée.

1.3.2.2 Développement de logiciels critiques sûrs

Le développement de système critiques en aéronautique est généralement réalisé en conformité avec les normes suivantes, qui couvrent différents niveaux :

- **Système** :
  - ARP 4754A/ED-79 (*Guidelines for Development of Civil Aircraft and Systems*) : processus de développement des systèmes avioniques, en intégrant des études pour permettre leur certification,
  - ARP 4761/ED-135 (*Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*) : ensemble de méthodes d'analyse et d'évaluation de la sécurité-innocuité des systèmes avioniques à des fins de certification,
- **Système IMA** : DO-297/ED-124 (*Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*) : contraintes de développement spécifiques à l'IMA, définit notamment la notion de partitionnement robuste,
- **Logiciel** : DO-178C/ED-12C (*Software Considerations in Airborne Systems and Equipment Certification*) : contraintes de développement liées à l'obtention de la certification de logiciel avionique,
- **Matériel** : DO-254/ED-80 (*Design Assurance Guidance for Airborne Electronic Hardware*) : contraintes de développement liées à l'obtention de la certification de matériel électronique avionique.

En particulier, l'ARP4761 définit différents niveaux de criticité applicables aux systèmes avioniques (en anglais, *catastrophic*, *hazardous*, *major*, et *minor*). Ce niveau est attribué à chaque composant après analyse des défaillances potentielles

du système et de ses conséquences sur le vol. A chaque niveau est associé une probabilité qu'une défaillance ne survienne par heure de vol. Ces exigences sont transcrites au travers des niveaux de DAL (*Design Assurance Level*) qui sont au nombre de cinq. Le niveau DAL A est le plus critique, et impose donc plus de vérifications/recommandations que le niveau DAL E qui est le moins critique.

Un système dont le dysfonctionnement peut provoquer un problème catastrophique impactant la sécurité du vol ou de l'atterrissage est classé DAL A. La probabilité d'un tel évènement ne doit pas dépasser 1×10^{-9} par heure de vol. En comparaison, le dysfonctionnement d'un système de niveau DAL E ne devrait pas avoir d'effet sur la sécurité du vol et aucun seuil de probabilité d'occurrence n'est donné.

Les exigences n'étant pas les mêmes selon le niveau de DAL, le développement ou la mise à jour d'un système DAL A coûtera beaucoup plus cher que l'équivalent sur un système DAL D ou DAL E.

1.3.2.3 L'aspect temps-réel

Les niveaux de criticité impliquent également des exigences en terme de temps de réponse des fonctions avioniques. Par exemple, il est primordial que le pilote connaisse sa position précise lorsqu'il est en phase de décollage ou d'atterrissage. De même, les calculateurs ne doivent pas induire de délai entre les actions du pilote sur les commandes et le mouvement correspondant effectué par les actionneurs. Ces temps de réponse sont notamment garantis par la ségrégation temporelle telle que décrite dans la Section 1.2.2. Dans un contexte IMA, quelque soit l'état des partitions s'exécutant sur le calculateur, le temps de calcul alloué à chaque partition et l'ordre d'exécution de l'ensemble sont garantis.

L'ajout d'une mesure de sécurité dans ce contexte doit prendre en compte cette exigence pour être insérée au bon niveau. Plusieurs niveaux peuvent être considérés pour insérer une mesure de sécurité dans ce contexte. Tout d'abord, le développement d'une mesure de sécurité dans une partition (applicative ou système) permet de profiter de la flexibilité offerte par l'IMA, mais ne permet pas d'observation fine de l'exécution de l'ensemble des partitions (la partition de sécurité ne sera capable d'observer l'état du calculateur et donc des autres partitions seulement pendant sa propre exécution). Ensuite, une observation continue des partitions présentes sur le calculateur peut être réalisée au travers d'une modification du noyau temps-réel (RTOS). Cependant, un RTOS capable d'héberger des partitions de niveau DAL A doit être lui-même de niveau DAL A, ce qui implique un coût très important pour une modification. Au-delà du coût de développement d'un RTOS de niveau DAL A, la modification du RTOS peut avoir un impact important sur l'exécution temps-réel des partitions hébergées, par exemple si le temps de réponse aux appels système est modifié. Enfin, le développement de mesures de sécurité au niveau hardware est également envisageable, et présente des avantages et inconvénients similaires au développement de mesures de sécurité directement dans le RTOS.

1.4 Objectifs de la thèse

Le modèle de menace exprimé dans cette thèse est celui d'une application malveillante parvenant à s'exécuter sur un calculateur avionique. Plus précisément, les risques considérés dans ce scénario sont d'une part la corruption d'une autre partition applicative, et d'autre part une déviation de comportement de l'application corrompue. Dans le cadre des avions de transports civils, la mise en place de mesures de sécurité cherche à maintenir la sûreté du vol, on parle donc de *Security for Safety*. Les besoins en sécurité se concentrent donc sur la disponibilité et l'intégrité des fonctions avioniques. Dans le cadre des avions militaires, la confidentialité de données telles que les données de mission doit également être préservée.

Si plusieurs types de mesures de sécurité existent, la plupart se concentrent sur la menace d'un attaquant externe, et cherchent donc à empêcher une intrusion plutôt qu'à la détecter et la traiter. C'est pourquoi cette thèse se focalise sur les systèmes de détection d'intrusion appliqués aux données applicatives. Ce type de système a déjà démontré son efficacité dans d'autres domaines pour détecter des attaquants internes, et serait intéressant à déployer dans un contexte avionique. Dans notre cas, cet attaquant interne pourrait être un employé chargé de développer une partie d'une application avionique, ou un opérateur de maintenance malveillant.

Cette thèse traitera uniquement de détection, pas de prévention ni de résilience après alerte, même si des exigences particulières seront portées sur les résultats de détection afin d'anticiper des travaux futurs sur une réaction après alerte.

Pour résumer, l'objectif de cette thèse est donc d'explorer les possibilités de mise en place d'un système de détection d'intrusion pour des applications avioniques, qui réponde aux différents critères énoncés dans ce chapitre parmi lesquels l'efficacité, l'embarquabilité, la certification, la confiance, l'interprétabilité, et les exigences en terme de ressources.

Chapitre 2

État de l'art

Sommaire

Ce chapitre propose un aperçu de techniques mises en place sur les aéronefs actuels et de travaux récents permettant d'assurer la sécurité d'un système avionique à différents niveaux. Tout d'abord, les mécanismes de *safety* sont présentés ainsi que la façon dont ils permettent de se prémunir de certaines menaces, même s'ils ne sont pas conçus à des fins de *security*. Ensuite, différents travaux ont été menés, que ce soit pour protéger les systèmes avioniques de façon périmétrique (contre un attaquant externe), pendant la phase de développement, ou pour se prémunir d'un attaquant interne (défense en profondeur).

Les travaux de cette thèse s'inscrivent dans le cadre de la défense en profondeur, pour lequel très peu de travaux appliqués aux systèmes avioniques existent. En particulier, la seconde partie de ce chapitre se focalise sur les systèmes de détection d'intrusion. Cette partie propose de décortiquer les éléments nécessaires au développement d'un tel système, notamment les données d'entrée, les techniques de détection, et les méthodes d'évaluation, afin d'orienter au mieux les choix qui seront effectués pour développer un tel système, adapté au contexte avionique présenté au Chapitre 1.

Enfin, ce chapitre énonce les contributions de cette thèse, et la façon dont elles sont restituées à travers les différents chapitres.

2.1 La sécurité dans les systèmes avioniques

Les systèmes avioniques sont historiquement très sûrs, du fait du grand nombre de mécanismes mis en place afin d'assurer le bon fonctionnement des aéronefs, même en cas de défaillance. Dans un premier temps, cette section détaille certains de ces

mécanismes et en quoi ils contribuent à la sécurité des systèmes avioniques, ou peuvent être la cible de malveillances. Ensuite, certaines mesures spécifiques à la sécurité périmétrique seront présentées, telles que les mesures de sécurité physique dans les aéroports ou les mesures de sécurité logicielles développées sur les aéronefs les plus récents. Enfin, les mesures de sécurité spécifiques à la phase de développement des systèmes avioniques seront évoquées, suivies des mesures liées à la défense en profondeur.

2.1.1 Les mesures de sécurité-innocuité

Comme présenté dans la partie 1.1.1, la sûreté de fonctionnement définit quatre moyens pour traiter des fautes : la prévention des fautes, la tolérance aux fautes, l'élimination des fautes, et la prévision des fautes. Cette partie présente un aperçu des mesures existantes pour chacun de ces moyens, et la façon dont elles influencent la sécurité générale des systèmes avioniques.

2.1.1.1 Mesures de prévention des fautes

La prévention des fautes consiste à empêcher l'occurrence ou l'introduction de fautes. Plus précisément, elle porte sur les erreurs de développement et les erreurs opérationnelles. La prévention des fautes, en évitant un développement ou une utilisation incorrecte du système, améliore directement la sécurité des systèmes en se prémunissant d'un grand nombre de vulnérabilités potentielles.

Les plateformes IMA sont très intéressantes d'un point de vue de la sécurité, puisqu'elles doivent garantir une ségrégation forte entre les partitions qu'elles hébergent. Pour être certifiée, une telle plateforme doit exposer un niveau de DAL au moins égal au niveau de DAL le plus élevé des applications installées. Le fonctionnement d'une plateforme de niveau DAL A, qui peut donc héberger tout type d'application, peut être prouvé et certifié par l'utilisation de méthodes formelles. Dans l'étude [VanderLeest 2018], les auteurs mettent en avant la possibilité d'utiliser un micro-noyau pour développer une telle plateforme IMA. L'avantage d'un micro-noyau est que le code est suffisamment petit (<10.000 lignes de code) pour pouvoir prouver mathématiquement son fonctionnement, sans engendrer un coût démesuré. Ils mettent également en évidence l'intérêt d'une telle preuve pour la sécurité, l'utilisation de méthodes formelles étant le seul moyen d'atteindre le plus haut niveau de sécurité, tel que le niveau EAL7 des Critères Communs [ISO/IEC 15408-1 2009]. Historiquement, les micro-noyaux n'étaient généralement pas utilisés dans les développements avioniques, du fait notamment de l'architecture fédérée utilisée. Cependant, cette piste est intéressante pour le développement des futurs calculateurs avioniques.

Concernant le développement des applications avioniques, plusieurs travaux proposent des langages de programmation adaptés au développement de logiciel critique sûr. L'étude [Soulier 2015] donne un aperçu des principaux langages utilisés pour développer des systèmes cyber-physiques, la plupart du temps embarqués avec des

propriétés de sûreté élevées.

Parmi les langages les plus utilisés dans le domaine avionique, on retrouve le langage ADA [Taft 2006]. Ce langage fortement typé a été développé spécifiquement pour répondre au besoin de code sûr des systèmes embarqués, en proposant notamment des protections qui évitent par exemple l'accès à des zones mémoire non allouées, les débordements de piles, ou encore qui vérifient les valeurs des variables selon des intervalles de valeur prédéfinis. D'un point de vue *security*, l'utilisation d'un tel langage réduit drastiquement les vulnérabilités exploitables par un attaquant.

Le langage C est également très largement utilisé dans les systèmes embarqués, du fait des possibilités de gestion mémoire fine que ce langage offre. Même si ce langage est statiquement typé, ce qui est un avantage direct pour le développement de logiciel critique sûr, il présente néanmoins une gestion des types et une gestion de la mémoire peu sûr. Plusieurs travaux visant à éviter ce type d'erreurs lors du développement existent. Par exemple, certains dialectes tels que Cyclone [Jim 2002] ont été développés pour éviter les erreurs les plus communes dans le code C. Deputy [Condit 2007] propose une extension au langage C afin d'intégrer des contraintes sur des valeurs. Dans le domaine avionique, des compilateurs certifiés sont utilisés afin de durcir les règles d'utilisation du langage C en un sous-ensemble sûr.

Dans la chaîne d'outils relatifs au développement d'une application IMA, on retrouve également un outil de gestion de configuration [Butz 2007]. Cet outil permet à l'intégrateur de calculateur de vérifier automatiquement un grand nombre de règles relatives à l'allocation correcte des ressources d'un calculateur. Ces règles sont directement liées au calculateur utilisé, et sont certifiées. Elles représentent le domaine d'usage du calculateur. Si ces règles sont respectées, le bon fonctionnement du calculateur est garanti.

Toutes ces règles et ces outils permettent de développer des logiciels critiques très sûrs, et donc de réduire significativement les vulnérabilités exploitables plus tard par un attaquant. Cependant, la mise en place de ces mesures est en grande partie basée sur des processus humains (utilisation des outils). D'un point de vue *security*, on peut imaginer des scénarios d'attaque interne visant directement ces mécanismes, par exemple en modifiant les règles des outils de configuration, en n'utilisant pas le compilateur certifié, ou en modifiant les FLS sous leur forme binaire.

Parmi les évolutions récentes de ce type d'outils, on peut évoquer le partitionnement intrinsèque [Fumey 2018]. Son principe est de déléguer la vérification des règles d'intégration, directement à l'OS une fois l'application chargée. Cette mesure permet de conserver les propriétés décrites précédemment quant à la vérification du domaine d'usage du calculateur, tout en introduisant une vérification récurrente à chaque démarrage du calculateur. Cela améliore donc à la fois la sûreté de l'ensemble, mais aussi sa sécurité.

2.1.1.2 Mesures de tolérance aux fautes

La tolérance aux fautes consiste à fournir un service à même de remplir la fonction du système en dépit des fautes. Elle comporte la détection d'erreur, le blocage de la propagation de l'erreur, et le recouvrement de l'erreur.

La redondance est souvent utilisée comme mécanisme principal de la tolérance aux fautes pour améliorer la fiabilité en utilisant des ressources alternatives. Elle permet de détecter un équipement défectueux donnant un résultat erroné en comparant les résultats d'au moins deux équipements similaires. Cette mesure oblige l'attaquant à attaquer non pas un seul, mais l'ensemble des équipements dont il veut modifier le comportement, diminuant la vraisemblance d'une attaque sur un tel système. Cependant, si ces équipements sont identiques, l'effort supplémentaire à fournir par l'attaquant reste minime.

Une autre mesure, la diversification de la conception, consiste à faire développer une même fonction par des équipes cloisonnées. Pour attaquer un système basé sur de la diversification de la conception, un attaquant doit donc multiplier ses efforts pour développer une attaque similaire sur des fonctions développées différemment. L'effort est donc non seulement plus important (il faut trouver plus de vulnérabilités et vecteurs d'attaque), mais la possibilité d'exploiter les vulnérabilités pour obtenir un résultat similaire sur chaque fonction est aussi moins probable que s'il n'y avait qu'un seul équipement. Cependant, ce principe n'est employé que pour les systèmes les plus critiques du fait de son coût important. De plus, ces fonctions différentes peuvent s'appuyer sur les mêmes technologies ou intégrer des composants similaires (matériels ou logiciels) selon le niveau de dissemblance requis (fonctionnel ou physique), et peuvent donc contenir des vulnérabilités identiques [ANSSI 2012].

Dans les calculateurs IMA, les mesures de recouvrement sont implémentées à travers la fonction de *Health Monitoring* [Parkinson 2007]. Notamment, chaque type d'erreur est associé à un niveau de gestion d'erreur (processus, partition, calculateur), lui-même associé à une action de recouvrement (gestion de l'erreur par un processus spécifique, redémarrage ou arrêt de la partition, et redémarrage ou arrêt du calculateur). Ces associations sont déclarées dans des fichiers de configuration statiques gérés par le RTOS, qui ne sont pas modifiables lors de l'exécution.

Le processus de gestion de ce type d'erreur par le mécanisme de *Health Monitoring* est illustré par la Figure 2.1. Le niveau de l'erreur dépend du moment de son occurrence et des fichiers de configuration définis par trois différents acteurs :

— Le fournisseur de calculateur configure la sanction à prendre lorsque des erreurs apparaissent en dehors de l'exécution des partitions (arrêt ou redémarrage du calculateur).

— L'intégrateur calculateur définit le niveau des erreurs apparaissant lors de l'exécution d'une partition (niveau calculateur ou niveau partition), et configure les sanctions associées aux erreurs de niveau calculateur (redémarrage ou arrêt du calculateur).

— Le fournisseur d'application traite les erreurs apparaissant lors de l'exécution d'une partition et qui ne sont pas de niveau calculateur. Il définit le niveau

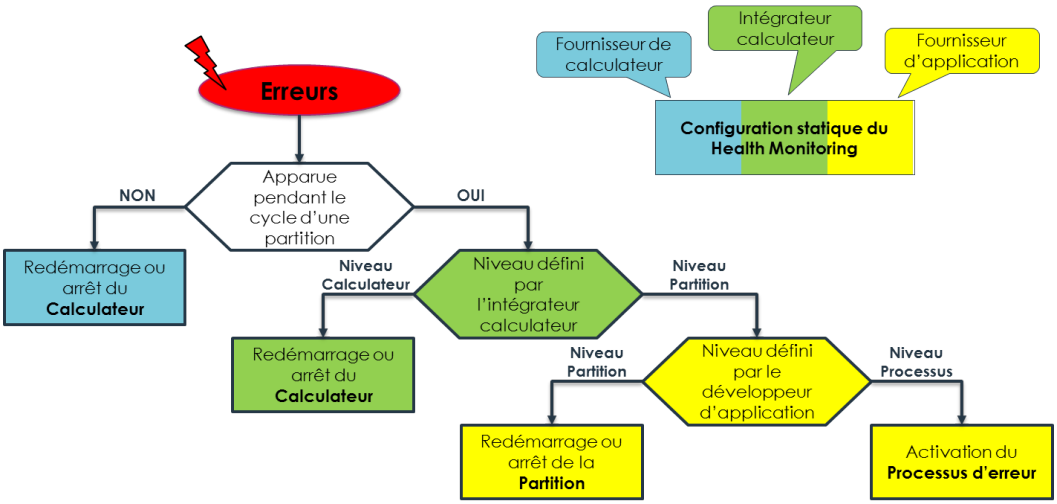


FIGURE 2.1 – Procédure de gestion des erreurs OS par le Health Monitoring

de l'erreur (niveau partition ou niveau processus) et les sanctions associées à chacun (redémarrage ou arrêt de la partition, ou activation du processus d'erreur).

Les mesures de détection d'erreur permettent de rendre plus difficiles les attaques sur les systèmes avioniques. Cependant, compte tenu des mesures de recouvrement associées, elles peuvent être elles-mêmes la cible d'attaque, notamment pour provoquer un déni de service. Dans la classification d'attaques sur les systèmes avioniques proposée dans [Dessiatnikoff 2013], une partie est consacrée spécifiquement aux attaques ciblant les mécanismes de tolérance aux fautes. Par exemple, une attaque peut cibler la façon dont les erreurs sont détectées en recherchant des cas spécifiques d'erronés non détectés avec des techniques de *fuzzing* et ainsi générer de possibles défaillances. D'autres attaques peuvent cibler les mécanismes de traitement des fautes pour tenter de l'activer sur des systèmes non défectueux, par exemple pour les isoler du système.

2.1.1.3 Mesures d'élimination des fautes

L'élimination des fautes consiste à vérifier si le système contient des fautes résiduelles après sa phase de conception et à les éliminer. Cette élimination de faute est faite à plusieurs niveaux lors du développement d'un aéronef : équipement, système, et avion. Cette vérification peut être effectuée de façon statique ou dynamique, selon la nécessité d'activer ou non le système. Le processus de développement des commandes de vol d'Airbus décrit dans [Bochot 2009] est repris par la Figure 2.2. Dans ce processus, des tests unitaires et d'intégration sont effectués au niveau équipement, des essais sur bancs de test sont effectués au niveau système, et des essais au sol sur des *iron bird* (représentatifs de l'ensemble des équipements avion) et des essais en vol sont réalisés au niveau avion.

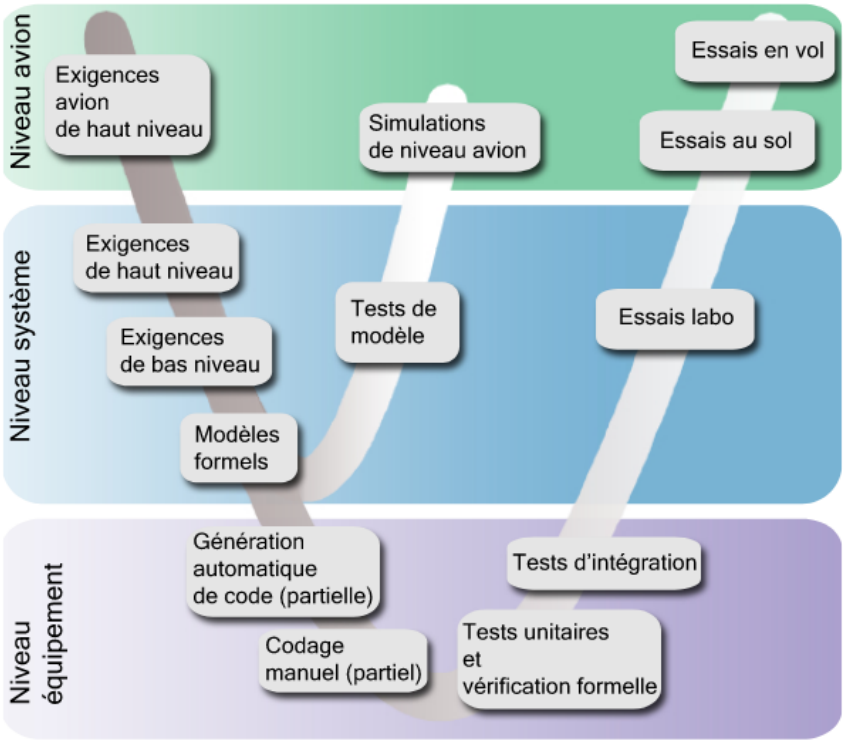


FIGURE 2.2 – Processus de développement des commandes de vol d'Airbus

L'ensemble de ces tests, comme les mesures de prévention de faute, participe à la suppression d'un grand nombre de vulnérabilités sur les équipements avioniques. Cependant, il est pratiquement infaisable de tester un système exhaustivement par rapport à toutes ses entrées possibles. Là où les tests de *safety* se concentrent généralement sur les fautes plausibles (définies d'après un modèle de fautes précis), un attaquant va provoquer intentionnellement un ensemble de fautes inattendues, qui n'auront potentiellement pas été prises en compte dans le modèle de fautes utilisé pour la *safety*. Cette menace est d'autant plus probable si l'on considère un attaquant interne : celui-ci pourrait avoir connaissance des documents retraçant l'ensemble des cas de test effectués.

Il faut également avoir à l'esprit que les tests les plus poussés sont effectués pour les fonctions les plus critiques. A contrario, il est beaucoup plus probable qu'une vulnérabilité existe sur un équipement peu critique. Ce type de vulnérabilité pourrait être exploitée par un attaquant afin d'atteindre une cible d'un niveau de criticité plus élevé. Cependant, ce type d'attaque nécessiterait également l'exploitation d'une autre vulnérabilité présente soit dans les mécanismes d'isolation de partitions, soit au niveau de la vérification des interfaces de la partition plus critique ciblée.

2.1.1.4 Mesures de prévision des fautes

La prévision des fautes concerne les méthodes et techniques destinées à estimer la présence, la création et les conséquences des erreurs. Elle est mise en œuvre à travers des analyses des risques d'erreurs liées au système considéré. En avionique, ces analyses sont appelées *Functional Hazard Assessment* (FHA), *Failure Mode Effect Summary* (FMES) et *System Safety Assessment* (SSA).

La FHA identifie les différentes situations redoutées, appelées *Failure Conditions* (FC). Celles-ci sont classées selon leur gravité. Des exigences sont associées à chaque niveau de gravité. Ces exigences se traduisent généralement par une probabilité maximale de défaillance par heure de vol sur le système concerné. La FMES liste les défaillances possibles des différents composants du système et donne leur taux de défaillance.

Finalement, la SSA doit démontrer que l'architecture finale du système est en accord avec les objectifs fixés. Elle vérifie notamment la cohérence entre les taux de défaillance des composants du système, donnés par la FMES, et les exigences décrites dans la FHA.

2.1.2 Les mesures de sécurité-immunité

Au-delà des mesures de *safety* présentées précédemment, de nombreuses initiatives relatives à la *security* dans l'aviation ont été lancées ces dernières décennies [Cooper 2017]. Cette section propose un aperçu de ces mesures selon trois axes : les mesures périmétriques, les mesures autour du processus de développement, et les mesures de défense en profondeur.

2.1.2.1 Les mesures de sécurité périmétrique

Le terme *périmétrique* se rapporte à l'ensemble des mesures permettant de former un périmètre de confiance autour d'un système. C'est par exemple l'objectif d'un firewall, qui doit protéger le réseau interne d'une entreprise du reste du réseau Internet. Dans l'aviation, on retrouve surtout des mesures attachées aux communications, qui doivent prendre en compte non seulement l'aéronef, mais aussi tout son écosystème : les aéroports et les services de gestion du trafic aérien ou *Air Traffic Management* (ATM). Parmi les initiatives récentes visant une amélioration de la sécurité des communications entre ces entités, un projet ARINC de modernisation du protocole de communication ACARS [SAE 2019] a été lancé en 2015. Son but est de remplacer le protocole ACARS, largement critiqué pour ses vulnérabilités, par un protocole basé sur IP (*Internet Protocol*) qui intègre les technologies actuelles de communication au sol, comme TCP, UDP ou IPv6. Des travaux récents ont également appliqué des mesures de sécurité sur l'architecture *L-band Digital Aeronautical Communications System* (LDACS) [Maurer 2019], qui pourrait remplacer progressivement les communications air-sol actuelles. Enfin, on peut évoquer la mise en place de mesures de vérification des FLS avant leur chargement et leur exécution dans la suite avionique [O'Neill 2016].

2.1.2.2 Les mesures de sécurité autour du processus de développement

Les mesures de sécurité autour du processus de développement concernent l'ensemble des mesures mises en place avant qu'un produit ne soit exploité en opération. Par exemple, [Dessiatnikoff 2013] propose une méthode d'analyse de vulnérabilités sur calculateur avionique à réaliser au cours de son développement, et permettant de supprimer un nombre important de vulnérabilités dans le produit final. Cette méthode a notamment permis de mettre en évidence et corriger plusieurs vulnérabilités sur un système embarqué expérimental développé par Airbus, utilisé comme cas d'étude dans [Dessiatnikoff 2013].

Ensuite, [Casals 2013] propose également une méthode d'analyse de risques qui s'applique spécifiquement aux systèmes avioniques. Ces travaux ont contribué à faire évoluer la réglementation autour de la sécurité des systèmes avionique, et notamment à la définition des standards ED-202/DO-326, ED-203/DO-356 et ED-204/DO-355, utilisés aujourd'hui.

Enfin, [Cooper 2017] propose un état des lieux récent concernant la sécurité de tout l'écosystème aéronautique, et met particulièrement en évidence la nécessité de réaliser des tests d'intrusions systématiques sur les différents composants avioniques afin d'écarter les menaces les plus critiques.

2.1.2.3 Les mesures de sécurité de défense en profondeur

L'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) définit le principe de la défense en profondeur comme suit : "La défense en profondeur consiste à protéger les installations en les entourant de plusieurs barrières de protection autonomes et successives. Elles peuvent être technologiques, liées à des procédures organisationnelles ou humaines." [ANSSI 2012]. Parmi ces mécanismes, on peut citer par exemple les mécanismes de durcissement des systèmes d'exploitation ou de détection d'intrusion. Le durcissement des systèmes d'exploitation permet par exemple d'éviter l'utilisation du système protégé comme point d'entrée ou rebond à l'intérieur d'un réseau protégé de façon périmétrique. La détection d'intrusion permet de surveiller en temps réel l'évolution de son système afin de réagir au plus vite en cas d'intrusion avérée.

Dans [O'Neill 2016], en plus de proposer un protocole de téléchargement de FLS sécurisé, l'implémentation d'un démarrage sécurisé du système d'exploitation est également proposée. Dans [Huyck 2019], les auteurs mettent également en avant des propriétés de durcissement de leur RTOS, leur permettant d'obtenir une certification de niveau DAL A et une certification EAL 7 sur un même RTOS. Leurs travaux récents ont étendu ces propriétés à des architectures multi-cœurs. Ils proposent notamment des contre-mesures visant à empêcher l'utilisation de canaux cachés entre des partitions.

Concernant les mesures de détection d'intrusion, très peu de travaux abordent le sujet spécifique des systèmes avioniques. Cependant, certaines études abordent l'introduction de tels systèmes sur des systèmes embarqués. Les travaux [Tabrizi 2015]

mettent par exemple en évidence les contraintes inhérentes à de tels systèmes, en proposant un système capable de s'adapter à des contraintes en terme de quantité mémoire disponible. [Studnia 2014] se focalise sur l'introduction d'un tel système sur des architectures réseaux automobiles et propose un système de détection d'intrusions sur le bus CAN permettant de détecter des attaques simples mais aussi des scénarios d'attaque plus complexes. L'utilisation d'une architecture multi-cœurs permettant de surveiller en temps réel l'exécution d'une application embarquée temps réel est explorée dans [Yoon 2013]. Dans cette architecture, un cœur est utilisé pour exécuter l'application tandis qu'un second cœur est dédié à sa surveillance.

Pour ces travaux, nous avons fait le choix d'explorer les mécanismes de défense en profondeur applicables au domaine avionique, et d'étudier plus particulièrement les techniques de détection d'intrusion pour les calculateurs avioniques. La section suivante présente donc un état de l'art spécifique aux systèmes de détection d'intrusion.

2.2 Les systèmes de détection d'intrusion

Les systèmes de détection d'intrusion, ou *Intrusion Detection System* (IDS) en anglais, sont largement utilisés dans les systèmes d'information. Afin d'intégrer un tel mécanisme sur un calculateur avionique, il est nécessaire de bien comprendre les différents types d'IDS existants, et de déterminer leurs principaux composants pour les reporter ou les adapter au contexte avionique.

2.2.1 Principe général et classification

De façon générale, les IDS se composent d'une entité de collecte de données et d'une entité de détection, qui peuvent s'exécuter simultanément ou de façon asynchrone. Dans la littérature, on différencie deux types d'IDS selon leur localisation dans le système d'information surveillé, et donc la nature des données collectées : les *Network-based* IDS (NIDS) et les *Host-based* IDS (HIDS). Les entités de collecte des NIDS sont placéss à des endroits stratégiques du réseau, par exemple sur un routeur dans un sous-réseau, ou sur un pare-feu à l'interface entre le réseau interne et le réseau Internet. Ce type d'IDS se base donc sur des données du réseau, tel que les adresses IP, les adresses MAC, la taille des paquets, leur fréquence, leur contenu, etc. On peut par exemple citer l'outil Snort [Snort 2019], qui effectue des analyses de paquets et des recherches de motif, ou encore l'outil IDIOT [Kumar 1995], qui fait également de la recherche de motif.

Les entités de collecte des HIDS sont placées directement sur des équipements terminaux du réseau, tels que des serveurs ou des stations de travail. Ils analysent par exemple des données applicatives, des fichiers, ou des données de journalisation. OSSEC [Hay 2008] est un HIDS en libre accès qui propose un très grand nombre de fonctionnalités : analyse de logs, vérification d'intégrité, surveillance des registres Windows, détection de rootkits, analyse temporelle d'évènements, et réaction après

alerte. On retrouve également un grand nombre de logiciels antivirus dans la catégorie des HIDS.

Les IDS peuvent également être différenciés en fonction de leur méthode de détection. Une classification commune consiste à différencier les IDS basés sur des signatures des IDS basés sur des anomalies. Le principe des IDS basés sur signature est de comparer les données courantes avec celles d'une base de signatures. Ces signatures correspondent à des motifs d'attaques déjà connues. Au contraire, les IDS basés sur des anomalies s'attachent à définir un comportement normal et à alerter lorsque le comportement observé dévie de façon trop importante par rapport à cette référence. Si les IDS basés sur des signatures ont été les premiers à se développer commercialement, des solutions d'IDS basés sur des anomalies émergent aujourd'hui, notamment grâce à l'essor des algorithmes d'intelligence artificielle (IA).

Ces deux techniques présentent des avantages et inconvénients résumés dans le Tableau 2.1. Au regard du domaine avionique, les IDS basés sur des signatures présentent des inconvénients majeurs. Tout d'abord, ce type d'IDS nécessite une base de données d'attaques connues, qui n'existe pas pour les systèmes avioniques aujourd'hui, à notre connaissance. Ensuite, la possibilité de ne pas détecter des attaques sophistiquées ou nouvelles, et le fait de devoir mettre à jour très régulièrement l'IDS semblent difficilement compatibles avec les exigences des systèmes avioniques formulées au Chapitre 1. Au contraire, les caractéristiques des IDS basés sur des anomalies semblent très intéressantes pour les systèmes avioniques, puisqu'un tel IDS serait capable de détecter efficacement des attaques pendant toute la durée de vie de l'aéronef, sans pour autant nécessiter de mise à jour. En effet, les applications avioniques sont très statiques d'une part, et leur code applicatif est très rarement mis à jour d'autre part (seules certaines bases de données sont mises à jour régulièrement). Cet aspect devrait faciliter la définition d'un modèle de comportement légitime pour une application avionique. L'ensemble des procédures autour du développement des systèmes avioniques semble également être un atout pour définir une phase d'apprentissage représentative des comportements attendus, et ainsi limiter les fausses alertes inhérentes à ce type d'IDS. Certaines études proposent de combiner ces deux approches pour tirer profit des avantages de chaque technique [Kim 2014, Om 2012].

Dans ses travaux, [Casals 2013] s'est focalisée sur le développement d'un NIDS avionique basé sur de la détection d'anomalie pour les raisons précédemment évoquées. La solution étudiée dans nos travaux se base également sur de la détection d'anomalie, en intégrant le principe des IDS hybrides. Nous avons fait le choix d'étudier l'intégration d'un IDS du côté applicatif, et donc les IDS de type HIDS. Ce niveau de détection est en effet plus adapté pour détecter des applications malveillantes au plus tôt, et représente une approche complémentaire à celle d'un NIDS avionique.

Les parties suivantes proposent un état de l'art relatif aux différents composants nécessaires pour définir une telle solution, à savoir les sources de données utilisables,

TABLE 2.1 – Principaux avantages et inconvénients des IDS basés sur des signatures ou des anomalies

	IDS basé sur des Signatures	IDS basé sur des Anomalies
Avantages	Rapide, précis	Nouvelles attaques et attaques avancées détectables
Limitations	Mises à jour régulières, nouvelles attaques indétectables, facilement contournable par des attaques avancées	Nombreuses erreurs (faux positifs ou faux négatifs), phase d'apprentissage difficilement exhaustive

les différentes techniques de détection d'anomalie pour traiter ces données, et les moyens permettant d'évaluer l'efficacité d'une solution d'HIDS.

2.2.2 Les données d'entrée

Le choix des données à observer est un élément clé pour la définition d'un HIDS, et d'autant plus pour un HIDS embarqué. D'une part, ces données doivent représenter au mieux le comportement de l'application à caractériser, afin de détecter au mieux ses comportements anormaux. D'autre part, il faut prendre en compte les contraintes de ressources et d'exécution des systèmes avioniques. On peut donc résumer les critères de choix d'une source de données de la façon suivante :

— Qualité de l'information : les données doivent porter suffisamment d'information pour être représentatives du comportement de l'application surveillée.

— Ressources allouées à la détection : la quantité de données générées ne doit pas excéder le budget mémoire alloué au HIDS, ni engendrer un dépassement de temps de traitement alloué au HIDS.

— Temps-réel : la collecte de données ne doit pas impacter de façon significative le fonctionnement du système avionique surveillé.

Les systèmes avioniques profitent déjà de plusieurs fonctions de surveillance, tel que le *Health Monitoring* évoqué dans la Partie 2.1.1.2. La réutilisation de ces systèmes a un double intérêt. Premièrement, cela permet d'envisager une implémentation de l'HIDS sur des aéronefs *retrofit*, c'est-à-dire qui sont déjà en circulation. Deuxièmement, cela n'implique aucun impact supplémentaire pour la collecte de données. Cependant, les données surveillées se limitent généralement à des occurrences de fautes ou des défaillances. Ces fautes incluent par exemple les codes d'erreur ARINC 653, des échéances ratées, des erreurs numériques, ou des requêtes illégales [Parkinson 2007]. Seul un sous-ensemble de ces fautes génèrent un message qui sera enregistré pour une investigation future par les opérateurs de maintenance. L'information portée uniquement par ces données risque donc d'être insuffisante pour détecter efficacement des intrusions. De plus, si l'application est corrompue, les messages d'erreur qu'elle génère pourraient être également corrompus.

Certains calculateurs proposent également des outils d'instrumentation qui peuvent être utilisés tout au long du processus de développement du système avionique (développement de l'application, intégration technique, intégration avion). Ils proposent des fonctionnalités permettant de vérifier la bonne utilisation des ressources, par exemple la quantité de mémoire utilisée, l'ordonnancement des processus et des partitions, l'utilisation de la mémoire non-volatile, ou les données transmises par les services de communication. Ce type de système est non intrusif et pourrait être une source de données très intéressante pour un HIDS. Cependant, les données surveillées et les outils associés ne sont pas standards (ils dépendent de la plateforme d'exécution), et les données restent limitées à l'utilisation des ressources.

D'autres sources de données peuvent être envisagées. Dans les travaux relatifs aux HIDS embarqués, les sources de données suivantes ont été explorées : l'utilisation de la mémoire [Yoon 2015b], la distribution des appels système [Yoon 2015a], la durée d'exécution [Yoon 2013], ou encore l'étude d'un sous-ensemble d'appels système [Tabrizi 2015].

Une étude a été menée en 2018 par Glass et al. [Glass-Vanderlan 2018] afin de proposer une classification d'HIDS en fonction des sources de données utilisées, sur des systèmes d'information classiques. Quatre catégories sont proposées, faisant chacune l'objet d'une section dans l'étude :

— Données d'audit et journaux système,

— Registres Windows,

— Surveillance du système de fichiers, et

— Binaires et processus.

Même si la surveillance des appels système fait partie de la catégorie "Binaires et processus", une section supplémentaire est dédiée aux HIDS utilisant cette source de données. Ces données d'entrée ont été largement utilisées, avec succès, pour détecter des anomalies dans plusieurs études. Dans la majorité des cas, ces appels système sont observés de façon séquentielle ou fréquentielle. Certaines études ont aussi examiné l'utilisation d'autres informations basées sur les appels système, comme les arguments utilisés ou les valeurs des pointeurs mémoire. L'observation des appels système est également le choix retenu dans [Kadar 2019], afin de développer un HIDS sur une plateforme temps-réel avec des applications de criticités multiples.

2.2.3 Les techniques de détection d'anomalie

Le choix de la technique de détection d'anomalie doit prendre en compte plusieurs critères tels que décrits dans le Chapitre 1. En particulier, on définit ici les quatre critères suivants, permettant de choisir une technique de détection d'anomalies adaptée au domaine avionique :

— Temps-réel : la technique utilisée doit être capable de traiter l'ensemble des données enregistrées pendant un cycle d'exécution de l'application surveillée, en moins de 5% de la durée totale du cycle.

— Empreinte mémoire : le modèle de comportement normal utilisé doit être suffisamment petit pour ne pas dépasser les exigences en terme de mémoire nécessaire.

— Résultats explicables : la définition du modèle et les anomalies relevées par rapport à ce modèle doivent être compréhensibles et interprétables.

— Applications boîte-noire : la technique utilisée doit être capable de fonctionner sans nécessiter une connaissance fonctionnelle de l'application sous surveillance.

2.2.3.1 La détection d'anomalie par apprentissage

La détection d'anomalies consiste à classer des observations selon deux classes (normale ou anormale). Les techniques d'apprentissage automatique proposent de nombreuses façons d'effectuer de la détection d'anomalie. Notamment, l'intérêt des techniques d'apprentissage automatique pour notre domaine est leur capacité à traiter des informations de toute nature, sans connaître explicitement le sens des données à traiter.

On peut citer deux principales techniques d'apprentissage automatique : l'apprentissage supervisé, et l'apprentissage non-supervisé [Russell 2010]. L'apprentissage supervisé consiste à guider l'apprentissage pour définir la frontière entre ce qui est normal et ce qui est anormal. Ce type de technique nécessite en particulier des données étiquetées des deux classes pour permettre ce guidage. Au contraire, l'apprentissage non-supervisé n'attache aucune connaissance aux données, et l'algorithme d'apprentissage doit définir lui-même différentes classes, ou déterminer si certaines observations sont trop différentes du reste des observations. Par la suite, des études ont proposé un troisième type d'apprentissage appelé apprentissage semi-supervisé. Ces techniques sont généralement utilisées lorsque certaines données sont étiquetées, mais pas la totalité.

Dans le contexte avionique, on peut prendre l'hypothèse que l'on sera capable d'exécuter une application en étant sûr que son exécution ne sera pas altérée. En d'autres termes, on considère que l'on est capable d'observer une application dans un environnement sain, et que l'on a une forte confiance dans cet environnement. Au contraire, il est plus difficile de s'assurer d'observer tous les cas d'attaque possibles sur une application avionique, et la prise en compte de telles données pour l'apprentissage pourrait être problématique. Les travaux présentés dans ce manuscrit se concentrent donc sur les techniques d'apprentissage semi-supervisé. En effet, ces techniques n'ont pas besoin d'exemples d'attaque pour apprendre un modèle de comportement normal, et détecter des anomalies par la suite.

2.2.3.2 Techniques applicables dans l'avionique

L'étude [Kwon 2017] propose une classification des techniques de détection d'anomalies à différents niveaux, qui se concentre particulièrement sur les modèles de type réseaux de neurones profonds. L'utilisation de ces modèles a montré une

grande efficacité, dans plusieurs domaines, pour détecter des anomalies. Cependant, beaucoup de ressources peuvent être nécessaires pour les traiter, et l'interprétation de leurs résultats présente aujourd'hui de réelles difficultés. Ces limitations semblent difficilement compatibles avec les exigences des systèmes avioniques, notamment celles relatives à la détection en temps-réel et à l'explicabilité des résultats.

D'autres techniques de classification ont été largement utilisées pour développer des IDS, comme citées dans les études [Chandola 2009] et [Buczak 2016]. Parmi ces techniques, le modèle *OneClass Support Vector Machine* (OCSVM) a été largement utilisé pour sa rapidité de détection. Dans [Casals 2013], il permet de modéliser des échanges au sein d'un réseau avionique ARINC 664 part 7 (*Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*). Dans [Maglaras 2016], il est utilisé pour modéliser des systèmes SCADA. Il est également utilisé dans [Daley 2016] pour détecter des anomalies dans des échanges sur un bus USB. Si un tel modèle semble compliqué à expliquer, sa rapidité de détection est un réel atout pour être utilisé dans un système avionique.

D'autres études citées dans [Glass-Vanderlan 2018] montrent que l'utilisation de chaines de Markov cachées, ou *Hidden Markov Model* (HMM) en anglais, pour modéliser l'usage des appels système, est très prometteuse. Les chaînes de Markov sans état caché sont facilement compréhensibles par un être humain, ce qui les rend très intéressantes pour le domaine avionique. Cependant, leur efficacité est vite limitée pour des systèmes complexes (par exemple, la prise en compte d'évènements rares mais normaux tels que les évènements relatifs à la sécurité-innocuité, dans un seul modèle), d'où l'introduction des états cachés dans les HMM. A contrario, l'introduction de ces états cachés rend plus difficile l'interprétation des résultats, pour effectuer un diagnostic par exemple.

Un modèle plus simple appelé automate temporisé a été utilisé dans [Liu 2017] pour détecter des anomalies dans le comportement d'un système de diffusion numérique de vidéos, et dans [Klerx 2014] pour détecter des fraudes sur un distributeur de billets automatique. Du fait du comportement très déterministe de ces deux systèmes, l'utilisation d'un automate temporisé a permis d'obtenir de très bons résultats de détection d'anomalies. Cette technique, très simple à expliquer et à interpréter, semble être un bon candidat pour détecter des anomalies sur des applications avioniques déterministes. De plus, cette représentation est parfaitement adaptée pour modéliser des séquences d'appels système et leur comportement temporel.

2.2.4 L'évaluation des HIDS

Cette section présente dans un premier temps les métriques utilisées pour évaluer des techniques de détection d'anomalie, puis présente différentes techniques d'évaluation des HIDS.

TABLE 2.2 – Matrice de confusion

	Positif réel	Négatif réel
Positif prédit	TP	FP
Négatif prédit	FN	TN

2.2.4.1 Métriques utilisées

La détection d'anomalies consiste à classer des observations comme normales ou anormales, selon un modèle donné. Dans le domaine de l'apprentissage automatique, cette classification consiste à prédire un label pour une observation, qui sera positif ou négatif selon la prédiction donnée par l'algorithme de détection. Dans ces travaux, les observations dites *Négatives* représentent des comportements normaux, tandis que les observations dites *Positives* représentent des anomalies. Une matrice de confusion, représentée par la Table 2.2, permet de caractériser les résultats prédits par rapport aux résultats attendus. Plus précisément, on retrouve les quatre catégories suivantes :

— Vrai Positif ou *True Positive* (TP) : l'observation d'une anomalie est prédite correctement comme une anomalie.

— Vrai Négatif ou *True Negative* (TN) : l'observation d'un comportement normal est prédite correctement comme un comportement normal.

— Faux Positif ou *False Positive* (FP) : l'observation d'un comportement normal est prédite de façon erronée comme une anomalie.

— Faux Négatif ou *False Negative* (FN) : l'observation d'une anomalie est prédite de façon erronée comme un comportement normal.

A partir de ces métriques de base, plusieurs métriques ont été développées pour représenter différentes caractéristiques de la détection d'anomalie. Les plus couramment utilisées sont l'exactitude ou *accuracy*, la précision ou *precision*, et le rappel ou *recall*.

L'exactitude représente le taux d'observations qui ont été classées correctement.

$$Exactitude : E = \frac{TP + TN}{TP + FN + FP + TN} \tag{2.1}$$

Elle permet d'avoir une vue générale de l'efficacité d'un détecteur d'anomalies, mais est très sensible au nombre d'observations de chaque classe (normale ou anormale). Par exemple, s'il y a beaucoup plus d'exemples négatifs réels, l'exactitude d'un système prédisant systématiquement les observations comme négatives pourra être excellente. Dans ce cas, il est important de vérifier la valeur d'autres métriques.

La précision représente le taux d'observations classées positives qui sont réellement positives.

$$Précision : P = \frac{TP}{TP + FP} \tag{2.2}$$

Elle permet notamment de vérifier l'absence de faux positif, donc d'observation normale classée comme anormale. L'absence de faux positif est une contrainte forte dans le domaine avionique, notamment pour construire un haut niveau de confiance dans un équipement. En effet, de fausses alertes pourraient avoir de graves conséquences pour une compagnie aérienne (avions cloués au sol, image de marque). A terme, si ces alertes sont suivies d'une réaction automatique directement dans les calculateurs, de fausses alertes pourraient également avoir un impact sur la sûreté du vol.

Enfin, le rappel représente le taux de positifs réels qui ont été correctement prédits comme positifs.

$$Rappel : R = \frac{TP}{TP + FN} \tag{2.3}$$

Cette métrique renseigne sur le taux d'anomalies qui ont été correctement classées comme telles. Généralement, les constructeurs d'IDS cherchent à maximiser ce taux, sans impacter trop fortement le nombre de faux positifs. En effet, les systèmes d'information classiques profitent de systèmes de supervision ou *Security Operations Center* (SOC) qui leur permettent de traiter chaque alerte par des opérateurs qualifiés. En avionique, même si la détection de tout évènement de sécurité est importante pour la sécurité du vol, on ne peut pas se permettre de traiter des faux positifs. En effet, un faux diagnostic peut mener à réagir d'une façon non conforme au problème à résoudre, et engendrer ensuite d'autres erreurs au sein de l'aéronef. Pour un système critique, il est donc très important de donner des résultats précis. Au-delà d'un impact potentiel sur la sûreté du vol, de fausses alertes peuvent impacter directement l'économie d'une compagnie aérienne, par exemple si celle-ci est obligée de clouer une flotte entière au sol par prévention. Les travaux présentés dans ce manuscrit se concentrent donc d'abord sur l'objectif d'atteindre un **taux de faux positifs nul**.

Ces métriques sont utilisées ici pour évaluer l'efficacité d'un HIDS applicatif. Dans ce cas, une observation positive correspond à une observation durant laquelle une attaque a été exécutée. Une observation négative représente quant à elle une observation durant laquelle l'application surveillée s'est exécutée de façon normale.

2.2.4.2 Techniques d'évaluation des HIDS

Les métriques présentées précédemment supposent toutes la manipulation d'observations anormales afin d'être évaluées. L'évaluation de l'efficacité d'un HIDS nécessite donc l'accès à des données d'attaque. Aujourd'hui, ces données peuvent être effectivement observées dans le cas de bases de données d'attaques utilisables, ou générées artificiellement. Afin de tester au mieux l'efficacité d'un HIDS, ces données d'attaques doivent être aussi exhaustives que possible. Le MITRE [MITRE 2018] propose en ce sens une classification d'attaques visant des systèmes informatiques traditionnels. Certaines classifications, comme [Gadelrab 2007], ont été proposées spécifiquement pour construire des campagnes de test pour évaluer l'efficacité de systèmes de détection d'intrusion. Cependant, peu d'études abordent les attaques

informatiques applicables aux systèmes avioniques et à leurs contraintes, comme la classification proposée dans [Dessiatnikoff 2012].

Généralement, la collecte de données réelles d'attaque s'avère complexe, d'autant plus pour des systèmes spécifiques comme les systèmes embarqués critiques. Plusieurs études proposent une alternative en focalisant sur des données générées expérimentalement, par exemple via l'utilisation de techniques d'injection de fautes [Arlat 1990]. Le but de ces injections est d'émuler des comportements malveillants représentatifs de la manifestation d'une attaque réelle sur le système ciblé. Par exemple, des outils de découverte de vulnérabilités tel que [Dessiatnikoff 2011] pourraient être utilisés afin de générer des comportements malveillants, mais également pour découvrir des vulnérabilités et définir des attaques réalistes en conséquence. Sans impacter la cible, l'outil ID2T [Vasilomanolakis 2016] injecte des paquets représentatifs d'une attaque dans une capture réseau effectuée sur le réseau réel d'une entreprise. Cet outil permet donc de créer artificiellement des données d'attaque représentatives de l'environnement, sans attaquer directement celui-ci. L'injection de vulnérabilités et d'attaques associées est également explorée dans [Fonseca 2009] sur des services web. Un module d'injection de vulnérabilité modifie le code du service web, tandis qu'un module d'injection d'attaque effectue des injections SQL sur ce service web pour exploiter la vulnérabilité introduite. Il n'y a pas à notre connaissance d'outil équivalent, permettant d'injecter du code malveillant, sur des applications avioniques.

Au contraire, les techniques d'injection de fautes ont été largement étudiées pour valider les mécanismes de tolérance aux fautes développés pour les systèmes critiques, dont les systèmes avioniques. Une étude de l'état de l'art sur l'injection de fautes par émulation logicielle a été proposée récemment dans [Natella 2016]. Cette étude présente en particulier les injections de mutations de code permettant d'émuler des fautes logicielles. Des mécanismes similaires pourraient être développés pour émuler du code malveillant. Il faut cependant prendre en compte quelques spécificités de l'injection d'attaques par rapport à l'injection de faute logicielle :

— Les instructions injectées dans le code binaire ne doivent pas forcément être représentatives d'erreurs réalisées dans le code source.

— La gestion du moment de l'exécution de la charge malveillante est directement géré par le code malveillant, et pas par un système extérieur. En ce sens, du code malveillant peut être exécuté sans activer sa charge malveillante directement. Il est cependant nécessaire de détecter cette exécution, avec ou sans activation de la charge malveillante.

— Les distributions de fautes proposées dans des études telles que [Duraes 2006] ne sont pas toujours directement applicables à des fautes intentionnelles.

En effet, [Duraes 2006] propose un ensemble de 18 opérateurs permettant de couvrir 68% des fautes logicielles les plus fréquentes. Ces opérateurs ont été revus et étendus par [Cerveira 2017] pour prendre en compte non pas les fautes (*safety*) mais les vulnérabilités (*security*) les plus fréquentes. Ces opérateurs ont permis de couvrir environ 57% des vulnérabilités relevées sur plusieurs projets open-source. Ils

ont également mis en évidence les difficultés à représenter des vulnérabilités à l'aide d'un seul opérateur. Ces opérateurs, s'ils permettent d'émuler des vulnérabilités logicielles, ne sont pas pour autant forcément représentatifs d'un morceau de code malveillant. L'émulation d'attaques, et d'autant plus sur des systèmes avioniques, reste donc un domaine de recherche peu exploré.

2.3 Contributions de cette thèse

Ce chapitre a présenté un aperçu des mécanismes existants ou en cours d'élaboration permettant d'améliorer la sécurité des systèmes avioniques. En particulier, les nombreux mécanismes de *safety* actuellement implémentés contribuent également à la *security* de systèmes avioniques à plusieurs niveaux : maîtrise de l'environnement, limitation des vulnérabilités potentielles, campagnes de tests, fiabilité des systèmes faces aux défaillances, etc. Cependant, ces mécanismes ne sont pas conçus spécifiquement pour se prémunir des malveillances, et ne sont donc pas suffisants pour assurer un bon niveau de *security* face à des acteurs malveillants, comme démontré par plusieurs incidents présentés dans le Chapitre 1.

A partir de ce constat, plusieurs études se sont focalisées sur la mise en place de mesures de *security* pour les systèmes aéronautiques. La plupart de ces mesures permettent d'assurer une sécurité périmétrique autour de systèmes avioniques, leur but étant d'empêcher toute intrusion depuis un domaine moins critique. Ces mesures se concentrent notamment sur les points d'accès aux systèmes avioniques, par exemple le protocole ACARS ou les systèmes de mise à jour de logiciels. Certaines études proposent également la prise en compte de la sécurité dans le développement de l'aéronef, afin de le rendre moins enclin aux attaques extérieures. Pour aller plus loin, des études se sont focalisées sur des mécanismes de défense en profondeur, où l'on considère que l'attaquant a réussi à contourner les mesures périmétriques en place. C'est notamment le cas de travaux sur le renforcement de la robustesse de l'OS pour faire des vérifications au démarrage, ou assurer une ségrégation forte entre applications. Néanmoins, il existe peu de travaux relatifs à la détection d'intrusion appliquée aux systèmes avioniques, qui sont donc à la fois critiques, temps-réel, et embarqués. A notre connaissance, une seule étude se focalise sur le domaine avionique à travers le développement d'un système de détection d'intrusion basé sur l'observation de données réseau.

Les travaux présentés dans cette thèse s'inscrivent donc dans les mesures de défense en profondeur applicables aux systèmes avioniques, et plus précisément les systèmes de détection d'intrusion permettant de détecter des applications avioniques malveillantes. A notre connaissance, il n'existe pas de travaux spécifiques à la mise en place d'une telle mesure pour des systèmes avioniques, qui utiliserait des données applicatives (et non des données réseau) comme source de données de surveillance. On parlera donc ici d'un HIDS avionique.

Même si de tels systèmes sont courants pour les architectures traditionnelles des systèmes d'information, l'adaptation d'une telle technologie au domaine avio-

nique n'est pas triviale, et il convient de bien comprendre le fonctionnement de ces systèmes traditionnels pour orienter au mieux nos travaux. Dans cette optique, ce chapitre a proposé un état de l'art des composants indispensables d'un HIDS, à savoir les données de surveillance, les techniques de détection d'anomalie, et les moyens d'évaluation des IDS. Cet état de l'art nous a également permis de constater que certains composants ne sont pas directement applicables à une architecture avionique. En particulier, il n'existe pas à notre connaissance de moyen efficace de tester l'efficacité d'un HIDS appliqué à des données applicatives avioniques. En effet, les moyens existants actuellement se concentrent généralement sur des systèmes d'information classiques, où le modèle de menace est très différent, et les attaques sont donc difficilement transposables.

Les contributions de cette thèse, par rapport à l'état de l'art présenté dans ce chapitre, sont les suivantes :

— Définition d'une approche permettant d'intégrer le développement, la validation, l'opération, et le maintien en conditions opérationnelles d'un HIDS avionique,

— Définition de la notion de *Domaine de Sécurité de l'Application* (SDA) comme étant un ensemble de règles représentant le comportement normal d'une application avionique,

— Proposition d'un processus de modélisation et de validation d'un SDA, basé sur 1) la proposition de différentes alternatives de SDA et 2) un outil d'injection d'attaque permettant d'évaluer l'efficacité d'un HIDS avionique et de calibrer au mieux ses paramètres,

— Implémentation d'un prototype d'HIDS embarqué dans un calculateur avionique et évaluation de sa consommation de ressources, et

— Proposition de pistes de réflexions quant à la mise en place d'un premier niveau de diagnostic en vol suite à la détection d'une intrusion.

Ces différentes contributions sont détaillées dans les chapitres suivants. Le Chapitre 3 propose une approche générale permettant de mettre en place un HIDS avionique, et présente ses différents éléments ainsi que la façon dont ils s'inscrivent dans un processus de développement avionique. La mise en place de cette approche a nécessité un travail particulier concernant les moyens permettant d'obtenir des exemples de comportements malveillants, donnant lieu à la définition d'un outil d'injection d'attaque dont le principe est également détaillé dans le Chapitre 3. Le Chapitre 4 présente l'implémentation concrète de cette approche sur un cas d'étude, et notamment l'implémentation de l'outil d'injection d'attaque. Plusieurs expérimentations sont également décrites, et permettent de mettre en évidence l'intérêt de l'approche proposée pour intégrer un HIDS avionique et la pertinence de l'outil d'injection d'attaque. Le Chapitre 5 se focalise sur la partie embarquée de l'HIDS. En particulier, plusieurs solutions alternatives ont été envisagées et évaluées en terme d'efficacité de détection. Dans ce cadre, deux solutions ont été implémentées de façon réaliste sur un calculateur avionique, afin d'évaluer concrètement leur

consommation de ressources. Pour finir, le Chapitre 6, plus prospectif, présente quelques réflexions sur la mise en place d'une confirmation d'attaque embarquée basée sur des signatures. Il propose notamment d'explorer les moyens de définir un premier niveau de diagnostic en vol suite à la détection de plusieurs anomalies, en s'appuyant sur des analyses au sol.

Chapitre 3

Approche générale de l'HIDS avionique

Sommaire

Ce chapitre propose une approche permettant de développer, calibrer, et opérer un HIDS embarqué dans un équipement avionique, afin de détecter une application malveillante. Dans un premier temps, une vue générale de l'approche est présentée, montrant la façon dont ses différents composants s'articulent entre eux pour couvrir les deux types de menaces considérés dans ces travaux. La notion de *Domaine de sécurité de l'application* est également introduite dans cette première partie. Dans un deuxième temps, chaque composant est détaillé avec l'implémentation envisagée pour chacun, et notamment les éléments nécessaires à leur mise en place. Parmi ces éléments, la définition d'un outil d'injection d'attaque a nécessité la réalisation de travaux particuliers qui sont détaillés dans un troisième temps. En effet, le manque d'exemples d'attaques réelles pour tester l'efficacité de notre approche nous a poussé à définir un tel outil afin d'émuler des comportements malveillants. Cet outil se base sur de la mutation de code à partir de trois stratégies distinctes. Enfin, ce chapitre se conclut par un retour sur les objectifs énoncés dans les chapitres précédents, et en quoi les différents composants de l'approche permettent d'y répondre.

3.1 Vue générale de l'approche

L'objectif de l'HIDS avionique proposé est de détecter une application malveillante qui s'exécuterait pendant l'opération d'un aéronef. En particulier, on considère que l'application a pu être soit développée de façon malveillante, soit corrompue après sa mise en production. L'approche d'HIDS proposée dans ces travaux cherche donc à couvrir les deux types de menace suivants :

1. Un FLS malveillant envoyé à l'intégrateur calculateur pour être intégré. Dans ce cas, nous prenons l'hypothèse que seul le binaire du FLS est corrompu, mais que la documentation et les entrées d'activation qui y sont rattachées sont correctes. Cette menace doit être détectée, au mieux à l'intégration, au pire lors de l'opération.
2. Un FLS est corrompu après la phase d'intégration, par exemple après une modification malveillante au sol, par une attaque externe, ou par une attaque provenant d'un autre équipement. Cette menace doit être détectée en phase d'opération.

L'HIDS que nous proposons dans ces travaux est basé sur la détection d'anomalies, et couvre donc principalement le deuxième type de menace. Les anomalies détectées sont des déviations du comportement légitime de l'application qui est obtenu par la définition du *Domaine de Sécurité de l'Application* ou SDA (*Security Domain of the Application*). Ce SDA représente le comportement attendu et légitime de l'application en absence d'intrusions. Ce concept est inspiré du Domaine d'Usage d'un calculateur avionique, qui représente un ensemble de règles permettant de garantir le bon fonctionnement du calculateur. Le SDA d'une application avionique est également un ensemble de règles qui caractérisent le comportement sain de l'application, et qui, si elles sont vérifiées en opération, garantissent que l'application n'est pas corrompue. Comme nous le verrons un peu plus loin dans ce chapitre, ce SDA peut prendre plusieurs formes mais on peut l'obtenir de façon relativement simple en profitant du comportement déterministe des applications avioniques, par exemple à partir de spécifications ou par l'expérimentation.

Afin d'adapter au mieux l'approche proposée au processus de développement IMA, celle-ci se base sur deux phases de ce processus : la phase d'intégration et la phase d'opération. Une vue générale des composants de cette approche est donnée dans une première partie, et montre la façon dont l'approche s'intègre dans le processus de développement IMA. La seconde partie présente le concept du SDA ainsi que les différentes formes qu'il peut prendre.

3.1.1 Composants de l'approche

L'approche est constituée de six composants présentés dans les Figures 3.1 et 3.2. Sur la Figure 3.1, les activités relatives à la phase d'intégration sont représentées. Sur la Figure 3.2, ce sont les activités relatives à la phase d'opération. Les activités représentées en vert sont réalisées au sol, tandis que les activités en bleu sont réalisées en vol.

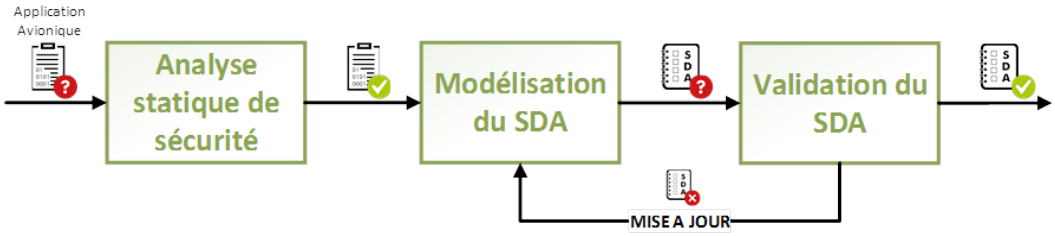


FIGURE 3.1 – Activités relatives à la phase d'intégration

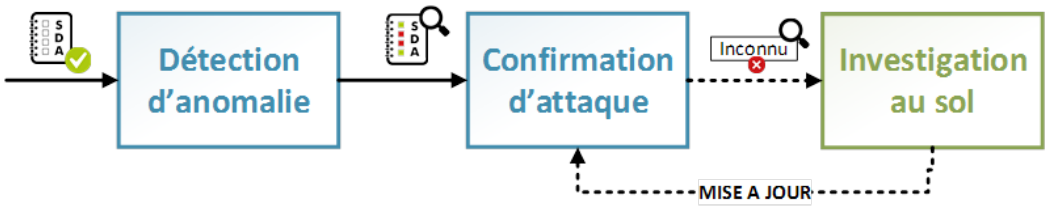


FIGURE 3.2 – Activités relatives à la phase d'opération

L'activité d'**Analyse statique de sécurité** permet de traiter le premier type de menace lors de l'intégration. Cette activité consiste à vérifier d'une part, que le binaire reçu ne contient pas de code malveillant connu (notamment, des virus visant des systèmes d'information plus classiques), et d'autre part, qu'il correspond bien à sa documentation (et en général, à toute information disponible sur cette application).

Les menaces détectables uniquement en opération (deuxième type de menace, et éventuellement premier type) sont traitées grâce à l'utilisation des deux activités embarquées pendant la phase d'opération : la **Détection d'anomalie** et la **Confirmation d'attaque**. L'activité de **Détection d'anomalie** est configurée au préalable, pendant la phase d'intégration, à l'aide du *Domaine de sécurité de l'application* (SDA). Les activités de **Modélisation du SDA** et de **Validation du SDA** permettent d'établir ce SDA à partir du binaire de l'application (vérifié au préalable lors de l'activité d'**Analyse de sécurité statique**) et de l'ensemble de ses entrées d'activation. On considère que ces entrées d'activation sont fournies par le développeur de l'application. Elles peuvent par exemple correspondre à l'ensemble des entrées de test utilisées pour la validation fonctionnelle de l'application.

D'après la littérature (voir au Chapitre 2), les techniques de détection d'anomalie nécessitent de faire un compromis entre le nombre de faux positifs et le taux de détection des attaques, ce qui se traduit généralement par un grand nombre de faux positifs. De plus, il est très difficile de garantir que le SDA représente bien l'ensemble des comportements normaux possibles de l'application surveillée (ceci est notamment dû au fait qu'il est généralement complexe d'obtenir des données d'entraînement exhaustives). L'objectif de l'activité de **Confirmation d'attaque** est justement de réduire ce nombre de fausses alertes, en utilisant une base de connaissances pour traiter les anomalies relevées. Les anomalies déjà rencontrées peuvent

ainsi être traitées directement, selon qu'elles représentent un faux positif, un comportement rare mais légitime, ou une attaque déjà identifiée comme telle. Si cette base de connaissances est incomplète et que des anomalies inconnues surviennent, une **Investigation au sol** est prévue, pour ajouter une nouvelle entrée dans la base de connaissances (ou la mettre à jour) une fois l'origine des anomalies établie. En particulier, cela permet de mutualiser les anomalies relevées sur un aéronef à toute une flotte.

3.1.2 Domaine de Sécurité de l'Application

Le concept de *Domaine de Sécurité de l'Application* (SDA) s'inspire du *Domaine d'Usage* défini pour un calculateur. Ce domaine d'usage définit un ensemble de règles qui, si elles sont respectées, garantissent le bon fonctionnement du calculateur. Ici, nous avons défini le SDA comme un ensemble de règles qui caractérisent le fonctionnement "normal" d'une application, c'est à dire le fonctionnement d'une application non corrompue. Par exemple, une règle peut correspondre à *"l'application ne doit pas faire plus de 100 appels API pendant un cycle d'exécution"*. Ces règles se basent sur l'utilisation des ressources du calculateur par l'application. Dans le contexte avionique, une application et son écosystème ne sont pas supposés évoluer dans le temps. Par conséquent, l'application devrait utiliser les ressources qui lui sont allouées de la même façon tout au long du cycle de vie de l'aéronef. En cas de mise à jour de l'application (qui est très rare pour la plupart des applications), le SDA évolue également en fonction de cette nouvelle version d'application.

Le SDA doit être adapté à la cible sur laquelle l'HIDS va s'exécuter, pour prendre en compte sa consommation de ressources (mémoire, CPU, bande passante), les éventuelles sources d'information déjà existantes (alertes *safety*, moyens d'instrumentation, informations de maintenance), et ses potentielles évolutions (le calculateur est-il en développement ou finalisé, peut-on effectuer des modifications). Pour cela, il est nécessaire de sélectionner un ensemble de paramètres pertinents pour définir le cadre du SDA. Dans l'idéal, il faudrait sélectionner le plus petit ensemble de données observables pour lequel la détection est efficace et adaptée à tout type d'application, pour un système donné. Plusieurs approches permettent d'explorer la sélection de ces paramètres, parmi lesquelles :

1. Sélectionner uniquement les informations déjà disponibles, par exemple pour être applicable à des aéronefs déjà en circulation,
2. Effectuer une analyse de risques et une analyse des effets des attaques sur un système, afin de sélectionner les informations les plus critiques,
3. Sélectionner les paramètres les plus communément surveillés par des HIDS dans la littérature (du domaine avionique ou d'autres domaines),
4. Sélectionner les informations les plus pertinentes après avoir expérimenté différents paramètres face à des attaques.

Dans notre cas, les approches 2 et 3 ont été utilisées pour orienter le choix des paramètres. Plus particulièrement, une analyse générale des risques sur une

TABLE 3.1 – Exemples de niveaux d'observation possibles du comportement d'une application avionique

Appels API	Lever un évènement à chaque fois qu'un appel API ARINC 653 est effectué par l'application
Code exécuté	Lever un évènement à chaque instruction exécutée par l'application
Communications	Lever un évènement à chaque fois qu'un message est envoyé ou reçu par l'application
Compteurs CPU	Lire les compteurs de performance du processeur
Mémoire	Lever un évènement à chaque fois que l'application fait un accès mémoire
Erreurs OS	Lever un évènement à chaque fois qu'une erreur est levée par l'OS

application avionique, et leurs effets sur le système, nous a permis de mettre en évidence six niveaux d'observations qui pourraient avoir un intérêt dans le domaine avionique, listés dans la Table 3.1.

Ces niveaux d'observation ne présentent pas les mêmes caractéristiques. Par exemple, il est probable que l'observation des *Compteurs CPU* ou des *Erreurs OS* soit plus facile à implémenter, puisque ces éléments sont déjà utilisés pour d'autres activités indépendantes de la sécurité (par exemple, pour évaluer les performances d'une application, ou pour la surveiller d'un point de vue *safety*). Au contraire, l'observation du *Code exécuté* semble beaucoup plus complexe à mettre en place, même s'il permettrait une observation très fine du comportement de l'application, qui serait très efficace dans un environnement statique. La surveillance des *Communications* permet de se concentrer sur l'utilisation des interfaces, qui représentent un accès privilégié pour un attaquant. L'observation des *Appels API* permet d'obtenir des informations assez variées sur l'exécution de l'application ou l'utilisation de ses interfaces, et a pour avantage de s'appuyer sur le standard ARINC 653. Par conséquent, le développement d'une solution sur un calculateur donné pourrait être plus facilement réutilisé sur un autre type de calculateur. Enfin, l'observation de la *Mémoire*, si elle semble assez complexe à mettre en place, permettrait toutefois de surveiller avec précision l'exécution d'une application, et serait très efficace pour repérer des attaques visant l'intégrité de l'application (et notamment des données qu'elle manipule).

Une fois le ou les niveau(x) d'observation choisi(s), il existe plusieurs façons d'observer les informations correspondantes. La Table 3.2 en propose quelques exemples, également basés sur notre analyse générale des risques sur une application avionique et de leurs effets sur le système.

Selon la quantité d'information observable, on peut envisager à minima de compter simplement le *Nombre* d'évènements, quel que soit le niveau d'observation. Si l'on est capable de différencier des évènements selon un niveau d'observation ou

TABLE 3.2 – Exemples de caractérisations possibles d'informations observées pour la modélisation du comportement d'une application avionique

Diversité	Nombre d'évènements différents ayant été levés
Nombre	Nombre d'évènements ayant été levés
Séquence	Séquences des évènements ayant été levés
Paramètres	Paramètres de l'évènement
Charge utile	Valeur manipulée lors de l'évènement
Horodatage	Horodatage de l'évènement
Type	Type de l'évènement, selon une typologie prédéfinie

selon un *Type*, il est envisageable de relever la *Diversité* d'évènements levés sur une période de temps donnée. L'observation de *Séquences* d'évènements nécessitera généralement un espace mémoire de stockage plus important mais donne des informations supplémentaires. L'observation des *Paramètres* d'un évènement est intéressant dans un contexte statique si l'on est capable de dénombrer les différents paramètres possibles pour un évènement. La *Charge utile* représente quant à elle des valeurs de paramètres avec plus de variabilité, comme des chaînes de caractères. Enfin, quelle que soit la connaissance sur un évènement, il est néanmoins possible de conserver des méta-données relatives à cet évènement telles qu'un *Horodatage*.

Ces tableaux n'ont pas vocation à proposer une classification exhaustive des informations pouvant entrer dans la définition du SDA, ni d'être efficaces de la même façon. Ils résultent d'une analyse générale et probablement incomplète, et cherchent avant tout à proposer des pistes quant à la sélection de paramètres à observer pour définir un HIDS avionique. Dans cette optique, chaque information surveillée devrait être composée d'un niveau d'observation et d'une caractéristique à observer. Par exemple, un SDA pourrait être composé de règles relatives au nombre d'appels API effectués par une application pendant un cycle d'exécution, au temps pour effectuer une séquence précise de communications, à la valeur des données lues ou écrites dans un segment mémoire spécifique, ou à la diversité des types d'instructions exécutées.

Dans nos travaux, nous avons décidé de nous concentrer sur la modélisation des appels API ARINC 653 effectués par l'application pour construire le SDA. En effet, de nombreux travaux ont démontré l'intérêt de l'observation des appels système pour effectuer de la détection d'intrusion (voir Chapitre 2), et cette observation peut facilement être réalisée au niveau du système d'exploitation. De plus, [Yoon 2013] a montré l'intérêt d'observer le temps d'exécution dans des systèmes temps-réel. Ces résultats sont notamment dûs au comportement périodique des applications observées. Dans notre contexte, cela peut se traduire par l'observation directe de l'horodatage, par le système d'exploitation, des appels API ARINC 653 effectués par l'application surveillée. Les travaux présentés dans ce manuscrit se sont donc concentrés sur l'observation des **types** et de l'**horodatage** des **appels API ARINC 653** effectués par l'application surveillée.

3.2 Description des composants

Cette section présente en détail les six composants de l'approche proposée. En particulier, l'implémentation envisagée pour chaque composant est donnée à titre d'exemple, afin de montrer comment cette approche s'adapte au contexte avionique.

3.2.1 Analyse de sécurité statique

L'objectif de l'analyse de sécurité statique est de détecter un FLS malveillant ou corrompu, que l'intégrateur a reçu. On prend l'hypothèse ici que seul le binaire peut avoir été corrompu, pas la documentation qui l'accompagne. Cette analyse se base sur deux idées principales :

1. Utiliser des techniques d'analyse anti-malware existantes sur le binaire, et
2. Vérifier la conformité entre le binaire de l'application et sa documentation.

Dans un premier temps, les techniques anti-malware existantes permettent de détecter la présence d'un code malveillant connu embarqué à l'intérieur du binaire de l'application. Par exemple, cette corruption a pu avoir lieu via l'utilisation d'une clé USB infectée, ou d'un environnement de production compromis. Même si le code malveillant n'est pas conçu pour viser spécifiquement un environnement avionique, cette détection peut permettre de trouver la cause de la corruption, de corriger la vulnérabilité, et de prévenir une future attaque ciblée utilisant ce canal. Cette approche ne semble pas spécifique au domaine avionique, et n'a donc pas été explorée dans ces travaux. De plus, la détection de code malveillant connu, et plus particulièrement l'analyse de code malveillant est un domaine de recherche très actif. Des études proposant un état des lieux en la matière sont publiées régulièrement [Jacob 2008, Gandotra 2014, Zennou 2018, Or-Meir 2019]. En particulier, il existe des recherches basées sur l'analyse morphologique de code permettant de reconnaître l'implémentation de certaines fonctionnalités, notamment celles utilisées de façon récurrente par les logiciels malveillants [Bonfante 2009, Bonfante 2017]. Si cette approche n'a pas été étudiée dans cette thèse, il serait néanmoins intéressant d'appliquer une telle technique sur du code avionique pour cette activité d'analyse de sécurité statique.

La deuxième approche cherche à tirer profit des contrôles déjà existants sur l'environnement de développement avionique. Plusieurs types de documents peuvent être envisagés pour effectuer une vérification du binaire, comme la spécification, le code source, une description, ou le contrat d'insertion. Dans la pratique, seul le contrat d'insertion est nécessairement connu de l'intégrateur calculateur, puisque c'est lui qui propose ce contrat au fournisseur d'application. Il contient des informations relatives à l'allocation d'espace mémoire, de temps CPU, de ports de communications, de services spécifiques, ou encore le nombre de partitions allouées.

Ces informations ne sont pas suffisantes pour caractériser précisément le comportement d'une application, et donc définir le SDA, mais peuvent donner un premier niveau de vérification. Par exemple, le contrat peut stipuler une allocation de 3Mo de mémoire non volatile, sans pour autant connaître la façon dont cet espace

sera utilisé (occasionnellement, périodiquement, très régulièrement, etc). Dans ces conditions, une sur-utilisation de cet espace mémoire ne pourrait pas être détectée avec la seule connaissance du contrat d'insertion. Cependant, le contrat d'insertion peut permettre de repérer des incohérences lors de la déclaration des ressources utilisées par l'application dans son code (par exemple, si 3 ports de communication sont alloués à l'application, mais que 4 ports sont déclarés dans son code).

3.2.2 Modélisation du SDA

L'activité de modélisation du SDA consiste à définir un modèle de comportement normal de l'application, selon des caractéristiques observables choisies au préalable. Comme précisé dans la Section 3.1.2, ces travaux se concentrent sur l'observation des **types** et de l'**horodatage** des **appels API ARINC 653** effectués par l'application surveillée pour définir le SDA. Cependant, l'approche est généralisable à d'autres caractéristiques observables. Cette section présente dans un premier temps le fonctionnement générale de l'activité de modélisation du SDA, avant de présenter plus en détail deux types de modèles utilisés dans ces travaux.

3.2.2.1 Présentation générale de l'activité

Les étapes de modélisation et de validation du SDA se réalisent de façon itérative et permettent de construire un SDA qui représente au mieux le comportement normal de l'application surveillée.

L'objectif de l'activité de modélisation du SDA est de proposer une version préliminaire du SDA, tel que décrit dans la Section 3.1.2. Il s'agit de construire un SDA au plus proche du comportement normal de l'application, pour différencier au mieux un comportement légitime d'un comportement illégitime. Le SDA peut être construit manuellement en utilisant la documentation disponible sur l'application, si elle est suffisante. Il peut également être construit de façon automatique en exécutant l'application dans un environnement de test. Dans ce cas, l'application est stimulée à l'aide de ses entrées d'activation. Une entité de surveillance observe simultanément le comportement de l'application, selon les paramètres choisis pour le SDA. Les entrées d'activation sont fournies par le développeur de l'application et doivent permettre d'activer l'ensemble des modes de fonctionnement de l'application. Ces entrées peuvent notamment correspondre aux entrées des tests fonctionnels réalisés pendant le développement de l'application. Par exemple, ces entrées d'activation peuvent être générées par une partition dédiée qui interagit avec l'application cible, ou à l'aide d'un ensemble de commandes réseau. La réutilisation des tests fonctionnels présente deux avantages principaux : limiter le sur-coût engendré par la modélisation du SDA, et permettre d'observer l'application dans l'ensemble de ses modes.

La Figure 3.3 propose un exemple d'implémentation pour l'activité de modélisation du SDA. Cet exemple comprend trois composants :

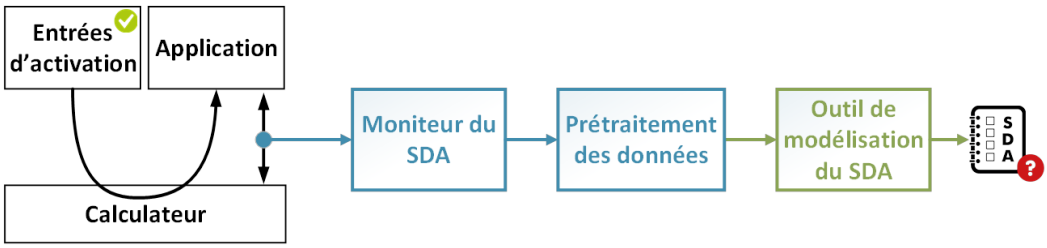


FIGURE 3.3 – Exemple d'implémentation pour l'activité de *Modélisation du SDA*

1. Le *moniteur de SDA*, qui observe le comportement de l'application et collecte les données correspondantes,
2. Le *prétraitement des données*, qui formate les données pour les rendre exploitables, et
3. L'*outil de modélisation du SDA*, qui définit automatiquement le modèle de comportement normal de l'application par une technique d'apprentissage semi-supervisé.

Dans cet exemple, l'application est stimulée à l'aide d'entrées d'activation provenant d'une partition dédiée.

Afin de développer l'outil de modélisation du SDA, il est nécessaire de choisir un type de modèle à utiliser. Concernant les choix relatifs au type de modèle, nous en avons sélectionné deux dans ces travaux : **OCSVM** (*One-Class Support Vector Machine*) et **Automate temporisé**.

3.2.2.2 1er choix de modèle : OCSVM

Dans un premier temps, nous nous sommes intéressés au modèle OCSVM, décrit dans [Scholkopf 2001]. Ce modèle est une extension du modèle SVM (*Support Vector Machine*), décrit dans [Vapnik 2000]. Le modèle SVM est conçu initialement pour effectuer de la classification supervisée entre deux classes, puis a été étendu pour la classification multi-classes et la détection d'anomalies. Pour la phase d'apprentissage, l'algorithme cherche un hyperplan optimal qui maximise la marge de séparation entre les deux classes tout en minimisant le nombre d'erreurs. Ce principe est illustré par la Figure 3.4 [1]. Les points des deux classes sont représentées respectivement en bleu et en rouge sur cette figure. L'algorithme d'apprentissage construit l'hyperplan décrit par la ligne pleine grise, tandis que les marges sont illustrées par les deux lignes pointillées.

L'un des atouts majeurs de ce modèle est sa capacité à s'adapter à des problèmes non linéaires, par application d'une fonction non linéaire sur les données, pour les projeter sur un espace de plus grande dimension. Cette fonction est appelée noyau ou *kernel*. Son application peut permettre de mieux classer des données, comme illustré sur les exemples de la Figure 3.5 [Gil Casals 2014]. Si l'on considère ces exemples, il serait impossible de classer correctement ces données de façon

1. https ://scikit-learn.org/stable/modules/svm.html

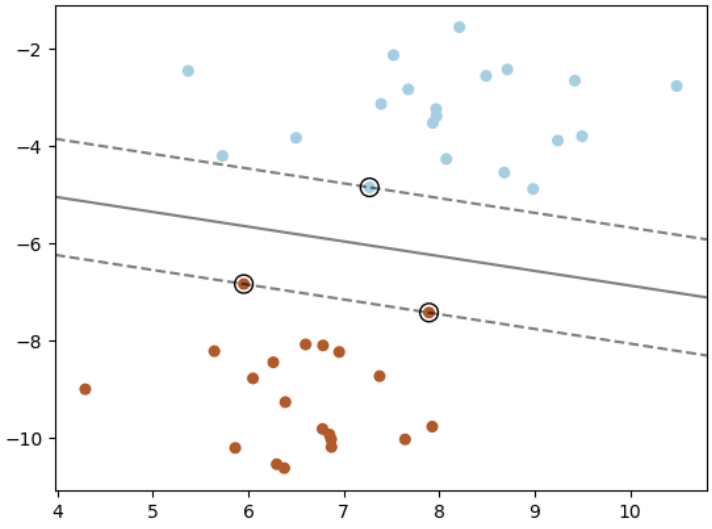


FIGURE 3.4 – Principe du SVM

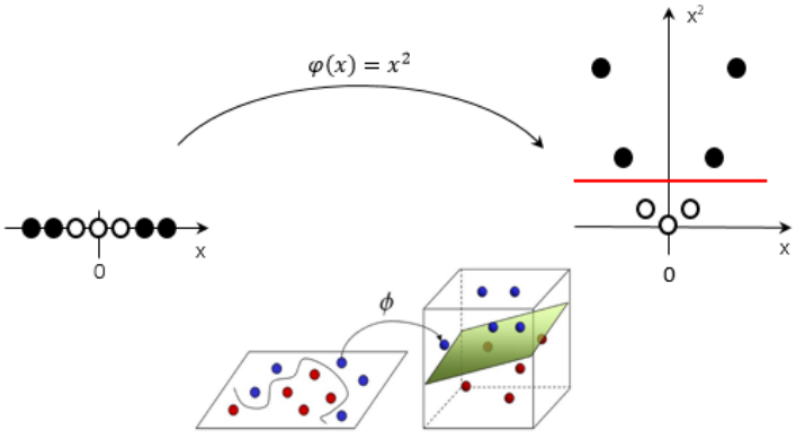


FIGURE 3.5 – Exemples d'utilisation d'un *kernel* avec SVM

linéaire. Cependant, avec l'application de la fonction kernel ($\varphi(x) = x^2$ pour le premier exemple, non précisée pour le second exemple), il devient alors possible de déterminer une séparation linéaire entre ces données.

Le modèle OCSVM est une extension du modèle SVM qui cherche à caractériser une frontière autour d'un ensemble de points normaux utilisés comme données d'entraînement. Ensuite, une fonction de détection permet d'évaluer un nouveau point comme étant similaire aux données utilisées pour l'entraînement (donc, à l'intérieur de la frontière), ou non (donc, à l'extérieur de la frontière). Cette fois, l'algorithme d'apprentissage proposé par [Scholkopf 2001] définit un hyperplan, non pas entre

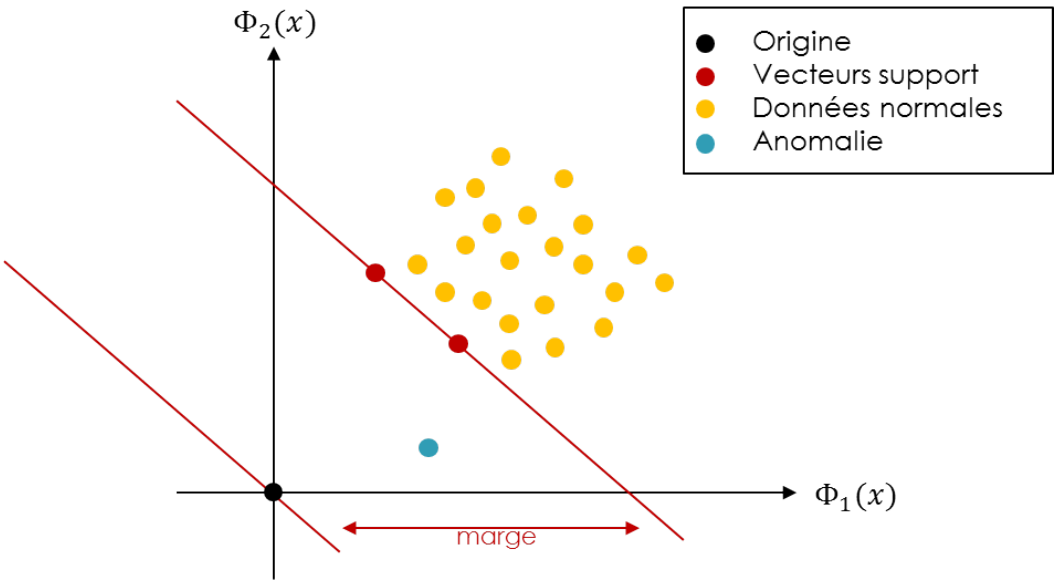
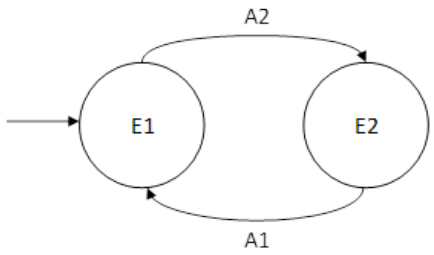


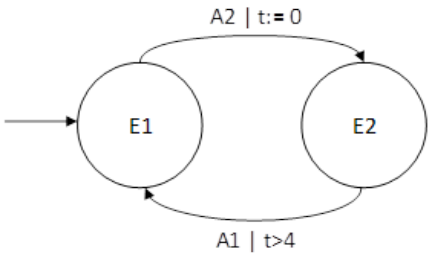
FIGURE 3.6 – Principe de l'OCSVM

deux classes (comme SVM), mais entre 1) les données d'entraînement et 2) l'origine du système de dimension $p \in \mathbb{N}$, tout en maximisant la distance de l'hyperplan à l'origine. Ce principe est illustré par la Figure 3.6. Le modèle OCSVM consiste en un ensemble de vecteurs support (représentés en rouge sur la figure) qui représentent l'hyperplan qui sépare les données d'entraînement (points jaunes sur la figure) de l'origine du système (point noir sur la figure). Lors de la détection, un point qui se situe trop proche de l'origine est considéré comme anormal (point bleu sur la figure).

Ce modèle OCSVM a été utilisé avec succès sur de nombreux cas d'application tel que décrit dans le Chapitre 2, et notamment dans les travaux de [Gil Casals 2014] pour introduire un NIDS comportemental au sein d'un réseau avionique. Plusieurs modèles d'apprentissage semi-supervisé sont capables de généraliser des données d'apprentissage et de démontrer des résultats similaires à l'OCSVM. Cependant, ce modèle est particulièrement intéressant dans notre contexte pour deux raisons. D'abord, il utilise peu d'hyper-paramètres (paramètres permettant de calibrer la phase d'apprentissage du modèle), ce qui lui confère une grande facilité de configuration. Ensuite, il présente de très bonnes performances en terme de rapidité de détection. Ce critère est très important dans notre approche, puisque c'est la partie de détection qui est embarquée, et donc exécutée en vol. Au contraire, optimiser le temps nécessaire à la phase d'apprentissage est moins important dans ce contexte, puisque cette phase est exécutée au sol, avec des ressources plus importantes à disposition.



(a) Exemple d'automate simple à deux états (E1, E2) et deux étiquettes (A1, A2)



(b) Exemple d'un automate temporisé à deux états (E1, E2), deux étiquettes (A1, A2), et une horloge (t)

FIGURE 3.7 – Représentation d'un automate et d'un automate temporisé

3.2.2.3 2ème choix de modèle : automate temporisé

Le second modèle étudié dans ces travaux est un automate temporisé. De façon générale, un automate est défini en informatique comme une machine capable de traiter de l'information de nature discrète (par exemple, des valeurs entières ou des caractères). Il permet notamment de représenter un ensemble d'états possibles pour un système et les transitions d'un état à l'autre. Un automate temporisé est un automate fini qui a la particularité de posséder une ou plusieurs horloges à valeurs réelles, qui augmentent toutes à la même vitesse. Ces horloges peuvent être ensuite utilisées pour autoriser ou interdire des transitions, en fonction de leurs valeurs. Cela permet d'imposer des contraintes de comportement à l'automate. Ces horloges peuvent être réinitialisées pendant l'exécution de l'automate. Les Figures 3.7a et 3.7b donnent une représentation d'un automate simple (Figure 3.7a) permettant de représenter des séquences d'appels API $A1$ et $A2$, et une représentation du même automate auquel une horloge t est ajoutée (Figure 3.7b). Dans cette seconde représentation, l'horloge est remise à zéro lorsque l'état $E2$ est atteint, et permet de définir une condition d'accès à l'état $E1$ depuis l'état $E2$ (i.e. l'appel $A1$ ne doit pas être exécuté avant que l'horloge n'ait atteint la valeur 4).

Le principal intérêt de ce modèle dans notre contexte est sa simplicité. D'une part, l'utilisation d'un modèle simple permet une bonne compréhension du modèle et des résultats de détection qui en découlent. Ce point est très important au regard des exigences de *safety* sur les systèmes avioniques. D'autre part, la simplicité du modèle induit une grande rapidité de détection et une représentation simple, qui permettraient de développer une solution embarquée efficace en terme d'utilisation des ressources (temps d'exécution et empreinte mémoire). Ensuite, le modèle d'automate est très adapté pour représenter des données séquentielles et a été largement utilisé pour représenter des séquences d'appels système. L'utilisation d'une telle technique a beaucoup d'intérêt pour des applications statiques, qui ne sont pas amenées à évoluer régulièrement dans le temps. Dans un automate dit "temporisé", la composante temporelle peut être très facilement représentée sur les transitions de l'automate. Ce modèle semble donc très adapté aux données que nous avons sélec-

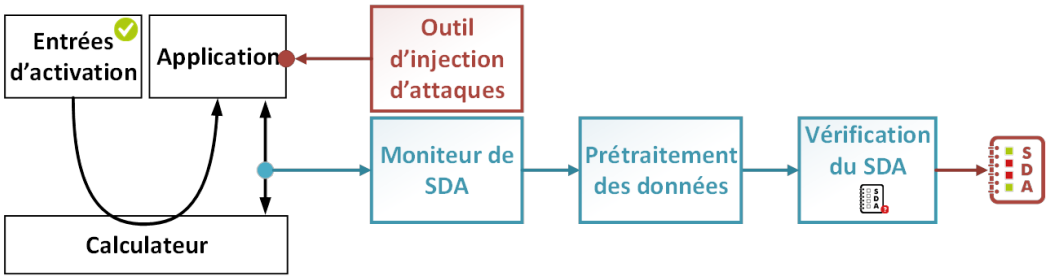


FIGURE 3.8 – Exemple d'implémentation pour l'activité de *Validation du SDA*

tionnées pour définir le SDA dans ces travaux. Enfin, si l'OCSVM permet une bonne généralisation des données d'apprentissage, il est intéressant dans ce contexte d'examiner un autre type de modèle tel qu'un automate, qui est beaucoup plus proche des données d'apprentissage.

3.2.3 Validation du SDA

L'activité de validation du SDA doit permettre d'évaluer si le SDA précédemment construit est efficace, c'est-à-dire s'il permet de distinguer correctement les comportements normaux des comportements anormaux. Pour cette phase, il est nécessaire d'avoir des exemples de comportement malveillant, et de sélectionner des métriques d'évaluation. Concernant les exemples de comportement malveillant, puisqu'il n'existe pas à notre connaissance d'exemple réel d'application avionique malveillante, nous proposons ici de les émuler à partir d'une étude des symptômes possibles d'une attaque. L'implémentation proposée pour l'activité de validation du SDA est illustrée dans la Figure 3.8.

Un *outil d'injection d'attaque* est introduit afin de modifier l'application, et ainsi générer des comportements anormaux. Par exemple, les attaques injectées peuvent simuler la désactivation du processus d'erreur en remplaçant son code par des instructions nulles (*nop*), remplacer un appel de fonction par un autre, ou introduire des boucles infinies à l'intérieur d'un processus pour bloquer l'exécution de la partition. Cet outil dédié est détaillé dans la Section 3.3. Il cherche à introduire des fautes dans le code de l'application surveillée à l'aide de techniques de mutation de code. Trois stratégies de mutation de code sont proposées afin de représenter la corruption d'une partie du code de l'application.

Concernant les composants *moniteur de SDA* et *prétraitement des données*, ce sont les mêmes composants que ceux utilisés pour la modélisation du SDA. La seule différence est que le comportement observé et formaté peut correspondre, soit à un comportement malveillant (si l'outil d'injection d'attaque est activé), soit à un comportement normal (si l'outil d'injection d'attaque n'est pas activé). Le dernier composant, la *vérification du SDA*, compare le comportement courant observé avec le SDA construit pendant la phase de modélisation du SDA. Il classe ensuite l'observation courante comme étant un comportement normal ou anormal. Sur un ensemble d'observations données, on est ainsi capable de donner un score au SDA,

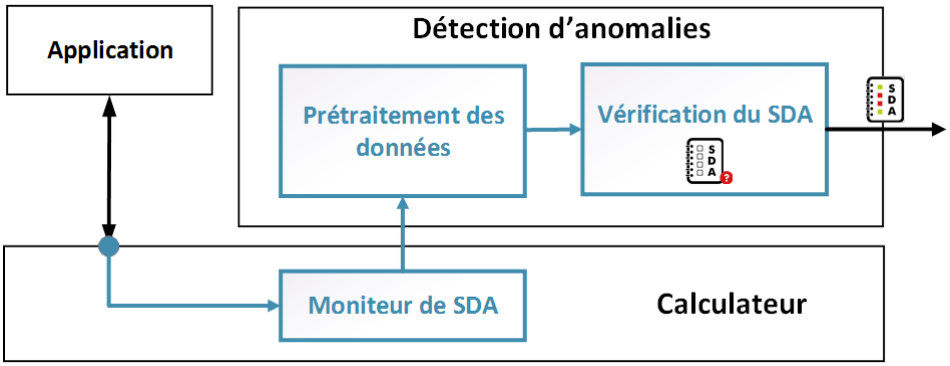


FIGURE 3.9 – Exemple d'implémentation pour la détection d'anomalie

en utilisant les métriques relatives à la détection d'anomalie telles que décrites dans la Section 2.2.4.1. Tant que le score obtenu n'est pas satisfaisant, les résultats sont étudiés afin de proposer une meilleure modélisation du SDA pour l'itération suivante. C'est seulement lorsque les résultats sont satisfaisants que le SDA est finalisé pour être utilisé dans les composants embarqués.

3.2.4 Détection d'anomalies

Une fois le SDA validé en intégration, il peut être déployé dans l'aéronef pour la phase opérationnelle, accompagné de l'application sous surveillance. Ce SDA permet de configurer la fonction de *Détection d'anomalies* embarquée pour une application donnée. Le rôle de cette fonction est de surveiller le comportement courant de chaque application installée sur le calculateur, et de le comparer en temps réel avec le SDA correspondant, de façon indépendante. Le comportement courant est considéré anormal par rapport au SDA si au moins un critère du SDA n'est pas vérifié. Chaque fois que le comportement courant n'est pas reconnu par le SDA, la fonction de détection d'anomalies relève une anomalie qui sera traitée par la fonction embarquée de confirmation d'attaque (décrite dans la section suivante).

La fonction de détection d'anomalies peut être embarquée à différents niveaux dans le calculateur, par exemple dans un matériel dédié, à l'intérieur du RTOS, ou dans une partition dédiée. Les possibilités d'ajout d'une mesure de sécurité à l'intérieur d'un calculateur ont déjà été évoquées dans la Section 1.3.2.3. Pour la fonction de détection d'anomalies, l'utilisation d'une partition système dédiée semble être un bon compromis. Dans ce cadre, la partition est développée directement par le fournisseur du RTOS. En plus de profiter des avantages d'une architecture IMA, la partition est développée par un acteur considéré comme de confiance dans ces travaux. De plus, l'interface avec le RTOS pour effectuer la collecte de données devrait en être facilitée.

C'est l'implémentation qui est proposée sur la Figure 3.9. Un composant de surveillance (le *Moniteur de SDA*) est directement ajouté au RTOS, et communique avec une partition dédiée chargée d'effectuer la comparaison entre le comportement

observé et le SDA, et d'enregistrer les anomalies relevées (*Prétraitement des données* et *Vérification du SDA*). Ces anomalies sont ensuite traitées par l'activité embarquée de confirmation d'attaque.

La fonction de détection d'anomalies ne doit en aucun cas interagir avec d'autres partitions, en dehors du cadre de leur surveillance. L'implémentation proposée ici permet de déporter la fonction de capture dans le RTOS, et ainsi éviter toute interaction directe entre la partition en charge de la détection d'anomalies, et les applications surveillées. En particulier, la fonction de détection d'anomalies n'a ainsi pas d'impact possible sur les applications critiques, et donc sur la sûreté du vol. Elle peut donc être développée avec un niveau de DAL faible. Cependant, il est important de garder à l'esprit que 1) la corruption de cette partition peut avoir un effet critique sur la sécurité du système, et 2) cette hypothèse sera à revoir selon l'utilisation des alertes levées par l'HIDS, notamment si l'on considère une réaction automatique dans le futur, ou un transfert de l'alerte à l'équipage.

3.2.5 Confirmation d'attaque et investigation au sol

Même si la détection d'anomalies présente de nombreux avantages pour le domaine avionique, le taux de faux positifs et le besoin potentiel de mettre à jour le SDA peuvent être problématiques. L'objectif de l'activité de confirmation d'attaque est à la fois de diminuer, voire supprimer, le nombre de faux positifs, mais aussi d'éviter la mise à jour d'un SDA. Pour réaliser cette tâche, trois fonctionnalités sont définies pour les activités de confirmation d'attaque et d'investigation au sol :

1. **Caractérisation des anomalies.** A l'aide d'une base de connaissance, la confirmation d'attaque doit caractériser les anomalies relevées par le détecteur d'anomalies, pour confirmer une attaque réelle, ou pour exclure un faux positif connu ou un ensemble d'anomalies liées à un évènement de *safety* qui serait déjà relevé par d'autres composants du système avionique.
2. **Envoi des alertes.** L'activité de confirmation d'attaque est responsable de l'envoi de l'alerte, une fois qu'une anomalie spécifique ou qu'une suite d'anomalies a été caractérisée. Le destinataire de l'alerte peut dépendre de la caractérisation effectuée. Par exemple, si une attaque est confirmée, on peut imaginer l'envoi d'une alerte directement au pilote avec une procédure à suivre. Si la fonction n'est pas capable de caractériser la ou les anomalie(s), l'envoi d'une alerte au sol, ou le simple stockage des informations relatives à cette caractérisation peuvent également être des alternatives envisageables.
3. **Mise à jour de la base de connaissances.** Lorsqu'une anomalie ou une suite d'anomalies n'a pas pu être caractérisée, l'approche prévoit une investigation au sol pour comprendre l'incident et le caractériser. Ce nouveau cas peut être ensuite inséré dans la base de connaissances de la fonction de confirmation d'attaque. Une telle mise à jour a deux avantages par rapport à la mise à jour du SDA : 1) la mise à jour se limite à l'ajout d'un cas dans la base de connaissances, ce qui devrait représenter une quantité de données plus faible

qu'un SDA complet, donc un temps de mise à jour plus faible, et 2) cette mise à jour est applicable plus facilement à une flotte entière d'aéronefs, et devrait être peu fréquente grâce à l'étape de validation du SDA faite à l'intégration, qui permet d'éviter un grand nombre de fausses alertes, voir de développer une première version de la base de connaissances.

L'implémentation d'une telle fonction peut être envisagée via une partition dédiée, voire un calculateur dédié qui traiterait l'ensemble des anomalies relevées sur les calculateurs du réseau. Dans ce cas, la consommation de ressources associées et les possibilités de certification doivent être considérées. Une autre approche plus simple à mettre en place serait de stocker l'ensemble ou une partie significative des anomalies relevées pendant le vol, et de les traiter au sol. Cette approche nécessite cependant une capacité de stockage plus importante. C'est néanmoins le choix qui est fait dans ces travaux, afin de proposer un premier niveau de traitement des alertes. C'est en particulier l'objet du Chapitre 6.

La section suivante décrit l'outil d'injection d'attaques que nous avons développé pour la phase de validation du SDA. Cet outil est fondamental dans notre approche générale et constitue en lui-même une contribution importante de nos travaux.

3.3 Définition de l'outil d'injection d'attaque

Dans un premier temps, cette section présente l'approche choisie pour développer l'outil d'injection d'attaque. La structure de l'outil et ses composants sont détaillés dans un second temps. La dernière partie de cette section est dédiée à la définition des opérateurs d'attaque, en se basant sur trois stratégies d'injection de code.

3.3.1 Vue d'ensemble

L'approche d'injection d'attaque proposée est très similaire à l'injection de fautes basée sur la mutation de code. Un modèle de faute, tel que décrit dans [Natella 2016], consiste à définir les trois points suivants :

1. Quand injecter la faute ?
2. Où injecter la faute ?
3. En quoi consiste la faute ?

Dans notre cas, nous nous intéressons spécifiquement aux attaques, qui sont des fautes intentionnelles malveillantes. Nous parlerons donc d'attaque plutôt que de faute dans la suite de ce chapitre. Le modèle d'attaque choisi pour cet outil est présenté dans cette partie.

Tout d'abord, la question **"Quand injecter l'attaque ?"** se rapporte au modèle de menace considéré. Dans ces travaux, pour les deux modèles de menace considérés (par exemple, un opérateur de maintenance malveillant ou un développeur d'application malveillant), on prend l'hypothèse que la modification malveillante dans le

FLS de l'application est réalisée **avant** que celui-ci ne soit chargé à bord de l'aéronef. On considère également que l'attaquant ne sera plus en mesure d'interagir avec l'application, une fois celle-ci chargée. L'attaquant peut donc chercher à dissimuler la charge malveillante, pour ne l'exécuter que selon des conditions précises, comme dans le cas d'une bombe logique. Par exemple, il peut s'agir d'attendre le décollage, l'atterrissage, le survol d'un océan, ou le passage au-dessus d'une zone sensible. Pour répondre à ce besoin spécifique, l'outil propose deux alternatives :

1. Exécuter la charge malveillante dès que l'application démarre, ou
2. Embarquer un code additionnel permettant de gérer l'activation de la charge malveillante selon des conditions spécifiques (temporelles ou événementielles).

La deuxième caractéristique du modèle d'attaque concerne la localisation de l'injection (**"Où injecter l'attaque ?"**). Dans notre cas, l'attaque doit être injectée à l'intérieur du FLS de l'application. Le code de l'application étant plutôt large, il est important de cibler des adresses spécifiques pour y injecter le code malveillant. Dans tous les cas, il y a de grandes chances pour que le code malveillant engendre un besoin supplémentaire de mémoire. Dans notre cas, nous choisissons de ne pas modifier la taille du FLS. Si un espace mémoire supplémentaire est nécessaire, il faudra donc identifier une zone de code à l'intérieur de l'application, permettant d'insérer un code additionnel. Par exemple, il est probable que l'ensemble du code de l'application ne soit pas activé lorsque celle-ci est stimulée avec un certain jeu d'entrées d'activation. Une zone de code non exécuté, pour un jeu d'entrées d'activation donné, peut donc être utilisée pour héberger cette partie de code supplémentaire.

Enfin, il est nécessaire de définir des types de mutation de code qui soient représentatifs de morceaux de code malveillants pour répondre à la question **"En quoi consiste l'attaque ?"**. Pour cette partie, trois stratégies de mutation de code sont proposées, afin de modifier le FLS de l'application et ainsi représenter l'ajout d'un code malveillant. A partir de ces stratégies, plusieurs opérateurs d'attaque associés ont été définis. Ces stratégies et ces opérateurs sont décrits en détails dans la Section 3.3.3.

3.3.2 Composants de l'outil

L'objectif de l'outil est de générer et exécuter automatiquement une campagne d'injection d'attaques adaptée à l'application sous surveillance. Pour le réaliser, l'outil comprend quatre composants : le scanner, le générateur, l'injecteur, et la configuration. Les interactions entre ces composants et l'application sont illustrées par la Figure 3.10. Les sous-parties suivantes détaillent chaque composant.

3.3.2.1 Scanner

Le *Scanner* utilise le **FLS** de l'application pour retrouver certaines informations sur l'application, qui permettront d'orienter le *Générateur*. Trois fonctions de scan sont implémentées dans l'outil, pour lister 1) les adresses des points d'entrée des fonctions, 2) les adresses des instructions de saut, et 3) les adresses des instructions

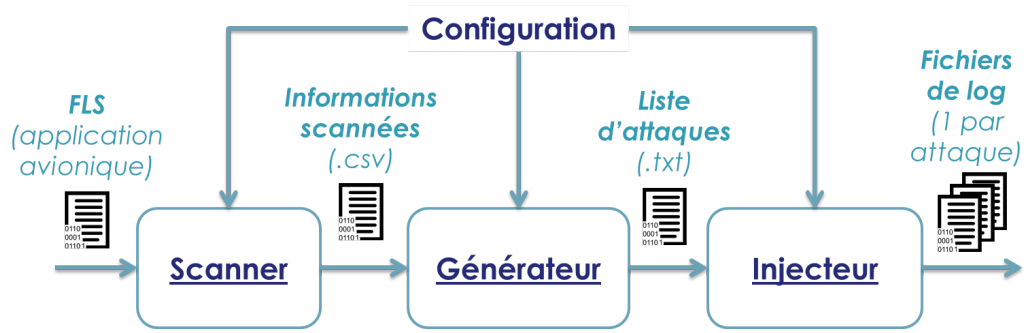


FIGURE 3.10 – Structure de l'outil d'injection d'attaque

exécutées selon un jeu d'entrées d'activation spécifique. Ces adresses sont stockées dans des fichiers *.csv*, et sont appelées les **Informations scannées**. Chaque fichier est associé à un type de paramètre, ici, 1) point d'entrée de fonction, 2) instruction de saut, et 3) instruction exécutée.

3.3.2.2 Générateur

L'objectif du *Générateur* est de générer une **Liste d'attaques**. Chaque ligne de cette liste d'attaques est composée d'un opérateur d'attaque et de paramètres associés. Selon le type de paramètre, le générateur va sélectionner une valeur aléatoire depuis les **Informations scannées**, ou une valeur aléatoire dans un intervalle de valeurs. Par exemple, la localisation de l'injection doit être sélectionnée parmi les adresses des instructions effectivement exécutées. Sinon, le code injecté ne sera jamais exécuté. Au contraire, une valeur de paramètre codée sur 4 octets peut être choisie aléatoirement entre 0 et $2^{32} - 1$. La liste générée est donc composée d'opérateurs d'attaque avec des paramètres aléatoires, qui seront différents d'une génération à l'autre.

3.3.2.3 Injecteur

Le rôle de l'*Injecteur* est de lancer automatiquement la campagne d'injection, selon une **Liste d'attaques**, et d'observer le comportement de l'application à chaque exécution. Pour chaque attaque de la liste, l'injecteur va 1) exécuter la commande d'injection d'attaque (l'opérateur et ses paramètres), 2) observer le comportement de l'application pendant une durée pré-déterminée, et 3) restaurer l'application dans son état initial et la redémarrer. Pendant chacune de ces exécutions, un fichier de log est utilisé pour recueillir les données d'observation de l'application. Une fois la campagne d'injection terminée, N **Fichiers de logs** ont donc été générés, chacun correspondant à une attaque de la liste utilisée en entrée.

3.3.2.4 Configuration

Le scanner, le générateur, et l'injecteur peuvent nécessiter des données de configuration, éventuellement partagées, qui sont représentées par le composant de configuration. Ces données de configuration peuvent être classées en trois niveaux.

Le premier niveau de configuration concerne les informations générales sur l'application ciblée, par exemple, les segments de code et données, le point d'entrée du binaire, ou l'adresse d'une zone de code non utilisée. Ces informations peuvent être fournies par l'intégrateur de calculateur, ou être retrouvées facilement par de l'analyse de code.

Le second niveau concerne les informations spécifiques à l'environnement, par exemple, la façon de restaurer et redémarrer le calculateur entre deux exécutions, la durée des exécutions, ou la façon de charger l'application sur le calculateur.

Le troisième niveau concerne les informations relatives aux attaques, par exemple, les opérateurs d'attaque à utiliser, les types de paramètres nécessaires pour chaque type d'opérateur d'attaque, ou les données d'observation à enregistrer pendant chaque exécution.

3.3.3 Définition des opérateurs d'attaque

Dans l'état de l'art présenté au Chapitre 2, deux études autour de l'injection de faute par mutation de code proposaient un ensemble d'opérateurs pour effectuer ces mutations. Ces opérateurs ont été définis pour représenter les fautes logicielles les plus fréquentes [Duraes 2006], ou les vulnérabilités logicielles les plus fréquentes [Cerveira 2017]. A notre connaissance, il n'existe pas d'étude équivalente pour définir des opérateurs représentatifs de code malveillant (opérateurs d'attaque). C'est ce que nous cherchons à définir à travers cette partie. En particulier, la définition de ces opérateurs d'attaque se base sur trois stratégies de mutation de code appelées *CrashMe*, *Substitution d'instruction(s) bien formée(s)*, et *Motif d'attaque*. Elles permettent d'effectuer des modifications de différents niveaux, que ce soit pour couvrir un large nombre de modifications possibles, ce qui peut entraîner des exécutions invalides, ou pour générer du code malveillant bien formé qui aura plus de chance d'être exécuté correctement.

Ces trois stratégies sont implémentées au travers des 12 types d'opérateurs d'attaques détaillés dans la Table 3.3. Ces opérateurs sont définis par une action (*Modification, Ajout ou Suppression*), appliquée à une cible (*Instruction, Saut, Appel de fonction, Registre, ou Valeur mémoire*). Certaines combinaisons ne sont pas représentées car elles n'ont pas toutes un sens (par exemple, on ne peut pas *Ajouter* un *Registre*).

3.3.3.1 Stratégie 1 : *CrashMe*

La stratégie *CrashMe* est la plus générique. Elle consiste à remplacer du code par un code aléatoire. Cette stratégie est inspirée du principe des attaques de type *CrashMe*, qui cherchent à exécuter du code aléatoire afin par exemple de

TABLE 3.3 – Types d'opérateurs d'attaque

Opérateur	Stratégie associée	Détail
Modification d'instruction aléatoire (MIA)	*CrashMe*	Remplace l'instruction courante par une valeur aléatoire
Modification d'instruction (MI)	*Substitution d'instruction(s) bien formée(s)*	Remplace l'instruction courante par une autre instruction
Modification de saut (MS)	*Motif d'attaque*	Remplace l'instruction de saut courante par une autre instruction de saut
Modification d'appel de fonction (MF)	*Motif d'attaque*	Remplace l'appel de fonction courant par un autre appel de fonction
Modification d'un registre (MR)	*Motif d'attaque*	Remplace la valeur d'un registre par une valeur aléatoire
Modification d'une valeur mémoire (MV)	*Motif d'attaque*	Remplace la valeur d'une cas mémoire par une valeur aléatoire
Ajout d'une instruction (AI)	*Motif d'attaque*	Insère une instruction
Ajout d'un saut (AS)	*Motif d'attaque*	Insère une instruction de saut
Ajout d'un appel de fonction (AF)	*Motif d'attaque*	Insère un appel de fonction
Suppression d'une instruction (SI)	*Motif d'attaque*	Remplace une instruction par un *NOP*
Suppression d'un saut (SS)	*Motif d'attaque*	Remplace une instruction de saut par un *NOP*
Suppression d'un appel de fonction (SF)	*Motif d'attaque*	Replace une instruction d'appel de fonction par un *NOP*

découvrir des vulnérabilités, ou provoquer des dénis de service, comme présenté dans [Dessiatnikoff 2012]. Cette stratégie est implémentée à travers l'opérateur d'attaque MIA (cf Table 3.3), qui remplace une instruction par une valeur aléatoire de la même taille que l'instruction remplacée. Pour l'architecture PowerPC utilisée dans ces travaux, une instruction est codée sur une taille fixe de 32 bits. Cet opérateur est facile à implémenter et est applicable à n'importe quel emplacement dans le code ciblé. Ce type de modification aléatoire permet de couvrir un très grand nombre de modifications possibles, notamment celles utilisant des instructions incorrectes. Par contre, il est plus difficile d'émuler des scénarios d'attaque tels que l'exfiltration ou la falsification de données. En général, cet opérateur va plutôt provoquer des défaillances dans l'application, donc des dénis de service. Les deux autres stratégies proposées dans l'outil cherchent à spécifier des mutations plus à même de provoquer d'autres cas de scénarios d'attaque.

3.3.3.2 Stratégie 2 : *Substitution d'instruction(s) bien formée(s)*

La seconde stratégie se base sur un dictionnaire d'instructions bien formées, construit au préalable. Ce dictionnaire peut par exemple être une concaténation du binaire de l'application cible, d'autres applications, et/ou du code de l'OS. Le principe de cette stratégie est de remplacer des instructions par du code bien formé, sélectionné dans ce dictionnaire. L'application de cette stratégie permet de garder une couverture assez large tout en limitant le nombre de cas de défaillances générées. Les mutations ne cherchent pas à représenter directement une modification malveillante du code, mais peuvent provoquer des modifications qui vont engendrer un comportement différent par rapport à l'application originale. Comme les instructions insérées sont bien formées, les applications mutantes ont plus de chance de s'exécuter sans provoquer de défaillance, qu'avec l'utilisation de la stratégie *CrashMe*. L'opérateur associé à cette stratégie est l'opérateur MI de la Table 3.3.

3.3.3.3 Stratégie 3 : *Motif d'attaque*

La dernière stratégie cherche à spécifier de façon plus précise un code malveillant. Son principe est de représenter les actions unitaires d'un attaquant qui chercherait à faire une modification malveillante d'une application avionique. Pour définir les opérateurs associés à cette stratégie, nous avons donc implémenté plusieurs scénarios d'attaque sur une application avionique générique (utilisée notamment comme exemple lors de formations au développement d'une application avionique), et extrait les opérations élémentaires réalisées pour développer ces scénarios. Plus précisément, ces scénarios cherchaient à modifier les communications (contenu, en-tête ou configuration), modifier les processus (période ou code exécuté), et exfiltrer des informations (emplacement de la configuration, ports de communication). Les actions unitaires extraites de ces scénarios ont été généralisées sous la forme d'une action (*Modification, Ajout ou Suppression*) appliquée à une cible (*Instruction, Saut, Appel de fonction, Registre, ou Valeur mémoire*). Ils ont donc permis de définir les 10

opérateurs d'attaque MS, MF, MR, MV, AI, AS, AF, SI, SS, SF de la Table 3.3. L'utilisation de ces opérateurs permet de générer des applications mutantes qui incluent du code représentatif d'un code malveillant. Cependant, l'implémentation de ces opérateurs est plus difficile que pour les deux stratégies précédentes. Il faudra par exemple spécifier les types de paramètres à utiliser pour chaque opérateur, et les intervalles de valeur appropriés pour chacun.

3.3.3.4 Conclusion

Les trois stratégies proposées ici nous ont permis de définir 12 types d'opérateurs d'attaque afin d'émuler spécifiquement du code malveillant. Ces trois stratégies ont pour but de représenter différents scénarios d'attaque, visant la disponibilité, l'intégrité, ou la confidentialité de l'application cible ou des données qu'elle manipule. Les stratégies proposées se complètent en proposant des modifications plus ou moins ciblées, qui seront plus adaptées pour représenter différents types de scénarios. En particulier, la stratégie *CrashMe* a de grandes chances de provoquer des attaques de type déni de service, tandis que la stratégie *Motif d'attaque* permettra d'émuler des comportements plus précis visant par exemple l'intégrité ou la confidentialité des données manipulées. La stratégie *Substitution d'instruction(s) bien formée(s)* propose quant à elle un niveau intermédiaire qui peut couvrir des scénarios non pris en compte par la stratégie *Motif d'attaque*, avec une probabilité beaucoup plus faible de provoquer des dénis de service que la stratégie *CrashMe*.

3.4 Conclusion

L'approche présentée dans ce chapitre vise à définir un HIDS qui soit adapté au contexte avionique. Cette approche se base sur deux périodes du cycle de vie d'une application, à savoir la phase d'intégration et la phase d'opération, afin de s'inscrire au mieux dans le processus de développement d'un aéronef. L'objectif est également de réutiliser au maximum les activités déjà réalisées, afin de minimiser les sur-coûts relatifs à la mise en place de l'HIDS.

Parmi les spécificités liées au contexte avionique citées dans le Chapitre 1, l'approche proposée ici cherche à répondre aux exigences suivantes :

1. **Efficacité de détection :** La première exigence concerne l'efficacité de l'HIDS, c'est-à-dire sa capacité à repérer les attaques de façon exacte. Cette exigence est réalisée grâce à la combinaison de deux procédés. Dans un premier temps, le procédé itératif de définition du SDA permet d'obtenir un modèle de comportement normal au plus proche du comportement réel de l'application, dès la phase d'intégration. Dans un second temps, l'activité de confirmation d'attaque et sa capacité de mise à jour offrent une certaine flexibilité pour améliorer les résultats de l'HIDS au cours de la phase d'opération. Elles permettent notamment de traiter à posteriori de potentiels cas particuliers de comportement normal qui n'auraient pas été pris en compte lors de la définition du SDA. Néanmoins, l'efficacité de détection de notre approche dépend

directement de son implémentation. Le Chapitre 4 se focalise sur cette exigence en proposant une implémentation de l'approche et de l'outil d'injection d'attaque, afin d'évaluer l'efficacité de l'approche sur un cas d'étude donné.

2. **Durée de vie :** L'approche proposée est adaptée à un système dont la durée de vie est très longue, puisque le modèle de comportement de l'application est défini uniquement sur son comportement normal et d'après l'ensemble de ses modes de fonctionnement, qui sont connus dans le cadre d'une application avionique critique. L'application et l'environnement n'étant pas amenés à changer, une définition précise du SDA pendant l'intégration peut permettre de répondre à cette exigence.
3. **Performances :** Les performances en terme de ressources utilisées dépendent entièrement des paramètres choisis pour le SDA, et de l'implémentation de la fonction de détection d'anomalies. L'intérêt de l'approche sur ce point est qu'elle est suffisamment générale pour être adaptée à différentes exigences en termes d'utilisation de ressources. Le choix des données à observer et la façon de les traiter est à choisir avec précautions pour remplir ses objectifs. Dans ce manuscrit, plusieurs possibilités ont été évoquées dans le Chapitre 2, et le choix d'une configuration est traité de façon plus approfondie dans le Chapitre 5.
4. **Impact temps-réel :** L'approche proposée permet de limiter grandement l'impact sur l'exécution temps-réel des applications. Le fait de déporter un maximum de fonctions dans une partition dédiée permet de profiter des propriétés de ségrégation spatiale et temporelle de l'IMA, et donc limiter l'impact d'un HIDS sur le fonctionnement des autres applications. Dans cette configuration, le seul impact réside dans le moniteur de SDA, qui est hébergé par le RTOS. Sur des systèmes existants, il est possible d'utiliser une source de données existante pour contourner ce problème. Sur de nouveaux systèmes, il sera nécessaire de recalculer le pire temps d'exécution, ou *Worst Case Execution Time* (WCET), associé à l'ajout du moniteur.
5. **Impact sur la sûreté :** De la même façon, l'utilisation d'une partition dédiée limite grandement les interactions possibles entre la partition surveillée et la partition d'HIDS. L'utilisation du RTOS pour collecter des données permet de garantir que l'application d'HIDS n'a aucune interaction ou dépendance directe avec d'autres applications. Par conséquent, elle ne peut pas avoir d'impact sur la sûreté de ces applications. En revanche, si un mécanisme de réaction est mis en place suite aux alertes de l'HIDS, ce ne sera plus le cas. Il faudra donc qualifier les alertes relevées par l'HIDS de façon à les rendre sûres.
6. **Certification :** Enfin, tant que l'HIDS n'induit pas de réaction, un faible niveau de DAL pourra lui être affecté. Seul le moniteur, hébergé dans le RTOS, devra donc garantir le même niveau de DAL que le RTOS, qui peut atteindre le niveau DAL A.

Pour conclure, ce chapitre a permis de donner une vision générale de l'approche qui est proposée et évaluée dans ces travaux. En particulier, nous avons défini un outil d'injection d'attaque capable de modifier une application afin de représenter des comportements malveillants embarqués directement dans le code d'une application avant son chargement sur l'aéronef. Si l'implémentation proposée pour cette approche permet de satisfaire les critères de *Durée de vie*, *Impact temps-réel*, *Impact sur la sûreté*, et *Certification* de façon théorique, il est néanmoins nécessaire d'évaluer les critères d'*Efficacité de détection* et de *Performances* de façon expérimentale. Le chapitre suivant s'attache donc à proposer une implémentation de l'approche permettant d'évaluer le critère d'*Efficacité de détection*, tandis que le Chapitre 5 se focalise sur le critère de *Performances*.

Chapitre 4

Implémentation de l'approche pour l'évaluation de l'efficacité de détection

Sommaire

Ce chapitre présente une première implémentation de l'approche, réalisée afin d'évaluer le critère d'*Efficacité de détection*. La première section de ce chapitre présente donc l'implémentation générale de l'approche réalisée, tandis que la deuxième section présente spécifiquement l'implémentation de l'outil d'injection d'attaque. Enfin, la dernière section présente différentes expérimentations réalisées à partir de ce prototype afin d'évaluer la pertinence de l'outil d'injection d'attaque et les capacité de l'HIDS à détecter des écarts de comportement (e.g. des comportements malveillants).

4.1 Implémentation de l'approche

Cette section décrit en détails le prototype réalisé afin de valider l'intérêt de l'approche proposée dans ces travaux. L'objectif est de proposer une première implémentation simple de cette approche, et de vérifier son intérêt sur un premier cas d'usage. Une première partie décrit l'environnement d'étude utilisé dans ces travaux, et comment le prototype s'intègre dans cet environnement. Chaque composant du prototype est ensuite décrit individuellement.

4.1.1 Environnement d'étude

L'implémentation de l'approche nécessite trois éléments principaux : un banc de test, une station de prototypage, et une station qualifiée. De façon générale, un banc de test est composé d'une cible et d'un contrôleur. Pour notre approche, la cible est un calculateur avionique exécutant l'application à surveiller, et le contrôleur est une machine capable d'interagir avec la cible à l'aide d'outils dédiés, qui sont développés par le fournisseur de la cible. Le contrôleur est utilisé pour capturer de façon représentative le comportement courant de l'application à surveiller. La station de prototypage permet d'effectuer des premiers tests de façon simple en s'affranchissant des contraintes liées à l'environnement du banc de test. C'est donc sur cette station que le processus itératif de modélisation et de validation du SDA sera effectué. Enfin, la station qualifiée est spécifique au contexte avionique et contient les outils qualifiés permettant notamment de compiler une application avionique ou vérifier sa configuration. Cette dernière station est utilisée pour développer la partition HIDS qui sera ensuite embarquée sur la cible.

Dans le cadre de ces travaux, les équipements suivants ont été utilisés comme banc de test, station de prototypage, et station qualifiée :

1. **Le banc de test :** L'architecture du banc de test manipulé est représentée par la Figure 4.1. Il est composé d'un contrôleur et d'une cible. Le contrôleur est un ordinateur classique sur lequel sont installés les outillages nécessaires pour interagir avec la cible. Il y a notamment un outil d'instrumentation, un client de debug GDB [1], et un outil Wireshark [2] de capture réseau, avec un plugin permettant de décoder les trames réseau avionique ARINC 664 part 7 (A664p7). La cible est un calculateur avionique avec une architecture PowerPC 32 bits et les composants logiciels suivants :
   - Un OS temps-réel auquel est attaché un serveur GDB, permettant d'arrêter le temps-réel localement au calculateur via des points d'arrêt,
   - Une ou plusieurs partitions systèmes, dont une partition permettant une instrumentation temps-réel et non-intrusive des partitions applicatives, et une partition de maintenance *BITE* (*Built-In Test Equipment*) qui agrège des logs destinés à la maintenance,

1. GNU Debugger : https ://www.gnu.org/software/gdb/
2. https ://www.wireshark.org/

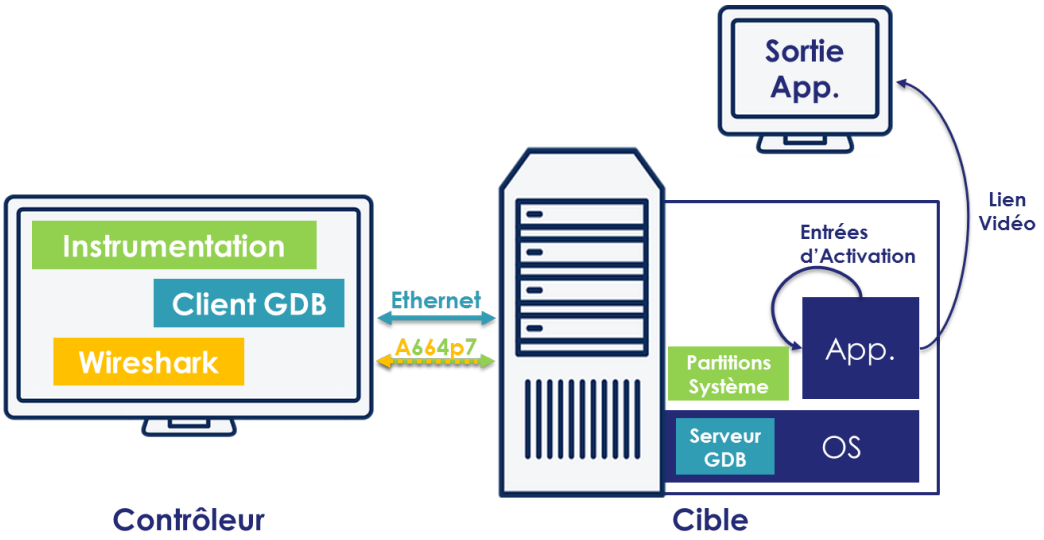


FIGURE 4.1 – Environnement banc de test

— Une ou plusieurs partitions applicatives, qui ne sont pas stimulées de façon extérieure (I/O ou interaction pilote), et qui peuvent éventuellement produire une sortie vidéo.

Le contrôleur et la cible interagissent via un réseau Ethernet et/ou via un réseau A664p7, selon l'outil utilisé.

2. **La station de prototypage :** C'est une station bureautique utilisée pour toutes les activités de prototypage. Au maximum, tout concept ou méthode est implémenté sur cette machine avant d'être implémenté sur la cible. Ceci a pour but de faciliter les développements et les tests de concepts ou technologies sur une machine rapide avant de passer dans un environnement embarqué plus difficile à appréhender.
3. **La station qualifiée :** C'est également une station bureautique, qui héberge cette fois l'ensemble des outils de la chaîne de développement d'une application avionique. Elle héberge notamment un compilateur certifié et des outils de gestion de configuration.

La Figure 4.2 décrit le prototype réalisé afin d'implémenter les activités de l'approche d'HIDS relatives à la phase d'intégration (Analyse de sécurité statique, Modélisation du SDA, et Validation du SDA). Ce prototype permet de travailler sur des données réelles (collectées depuis la cible) pour évaluer l'efficacité de détection de l'HIDS, en utilisant la station de prototypage. L'évaluation de l'utilisation des ressources par l'HIDS est traitée plus tard dans le Chapitre 5.

La cible utilisée est un calculateur en cours de développement dont le matériel est représentatif du matériel utilisé sur le produit final. Le RTOS est une version en développement également, dont la MMU n'est pas activée et qui ne possède pas de partition système d'instrumentation ou de maintenance (*BITE*). Deux applica-

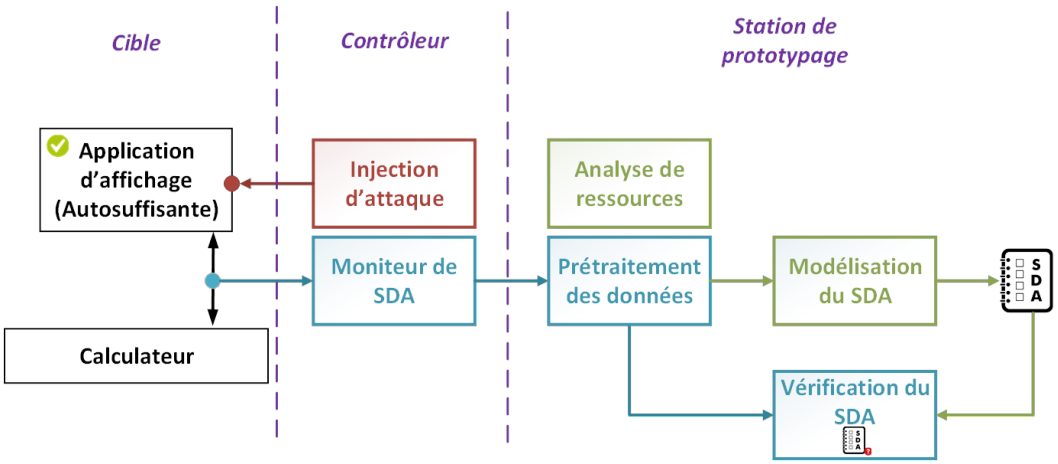


FIGURE 4.2 – Prototype relatif aux activités de la phase d'intégration

tions d'affichage appelées IHM-DA (Interface Homme-Machine pour les Données de l'Aéronef) et IHM-DV (Interface Homme-Machine pour les Données de Vol), en cours de développement, sont exécutées sur ce calculateur. Ces deux applications ne sont pas exécutées simultanément, il est nécessaire de sélectionner au préalable l'application qui sera exécutée. Elles affichent respectivement des informations relatives à l'état physique de l'aéronef, comme la pression des pneus ou la quantité de carburant disponible, et des informations relatives au vol, comme la vitesse ou l'altitude. Chacune de ces applications est développée sur une seule partition. Dans les deux cas, celle-ci procède à une phase d'initialisation au démarrage puis affiche des données relatives à un scénario codé en dur dans le code de la partition (ses entrées d'activation sont donc directement codées à l'intérieur de la partition, elles sont donc auto-suffisantes).

Deux nouveaux outils sont introduits sur le contrôleur : l'outil d'injection d'attaque et le moniteur de SDA. Le premier permet de modifier le comportement de l'application cible, et donc générer des comportements anormaux. Il est détaillé dans la Section 4.2. Le second permet d'observer le comportement de l'application sous surveillance et de stocker les données d'observation pour un traitement ultérieur sur la station de prototypage.

Cette station de prototypage héberge quatre nouveaux composants : un outil d'analyse de ressources, un outil de prétraitement des données, un outil de modélisation du SDA, et un outil de vérification du SDA. L'outil d'analyse de ressources est utilisé dans la phase d'**Analyse statique de sécurité**. Le moniteur de SDA et le prétraitement des données sont utilisés pendant les phases de **Modélisation du SDA** et de **Validation du SDA**. L'outil de modélisation du SDA est utilisé pendant la phase de **Modélisation du SDA**, et les outils d'injection d'attaque et de vérification du SDA sont utilisés pendant la phase de **Validation du SDA**.

Le Tableau 4.1 résume l'utilisation de ces composants dans l'approche HIDS lors de la phase d'intégration. L'implémentation de chaque composant est détaillée

TABLE 4.1 – Résumé des composants et de leur utilisation lors de la phase d'intégration

	Analyse statique de sécurité	Modélisation du SDA	Validation du SDA
Outil d'analyse de l'utilisation des ressources	×		
Moniteur de SDA		×	×
Prétraitement des données		×	×
Modélisation du SDA		×	
Outil d'injection d'attaques			×
Vérification du SDA			×

dans la suite de cette section.

4.1.2 Outil d'analyse de l'utilisation des ressources

Pour la phase d'analyse statique de sécurité, seul le concept de vérification de conformité entre le binaire d'une application et son contrat d'insertion a été implémenté sous la forme d'un outil statique d'analyse de l'utilisation des ressources. L'application de techniques anti-malware sur le binaire n'a pas été étudiée dans ces travaux.

Le principe de l'outil est de parcourir le code de l'application afin de calculer des métriques caractérisant l'utilisation des ressources, directement depuis le code binaire de l'application. L'outil se base sur la définition des services API ARINC 653. Ces services sont classés dans 6 catégories : gestion de la partition, gestion des processus, gestion du temps, communications inter-partition, communications intra-partition, et gestion des erreurs. Une septième catégorie, non standard, concerne les services de gestion de la mémoire non volatile.

4.1.2.1 Ressources considérées

Dans le cas d'étude utilisé ici, le contrat d'insertion contient les informations suivantes relatives à l'allocation de ressources de la partition IHM-DA, décrite dans la Section 4.1.1 :

- Quantité de mémoire RAM :
  - Allouée au code
  - Allouée aux données
  - Allouée aux communications inter-partitions
- Durée d'exécution
- Taille de la mémoire non volatile
- Nombre de ports de communication

— Services spécifiques autorisés
— Nombre de ressources physiques :
  — ARINC 429
  — Discrets (I/O)
— Nombre d'interfaces réseau :
  — A664p7
  — Ethernet

Si ces informations peuvent être vérifiées directement grâce aux outils de gestion de configuration, certaines informations peuvent également être vérifiées par analyse statique du code binaire : la quantité de mémoire RAM utilisée pour les communications inter-partitions, la mémoire non volatile utilisée, le nombre de ports de communication utilisés, et les services spécifiques utilisés.

Par exemple, la mémoire non volatile est déclarée dans le code de l'application à l'aide des services *CREATE_LOGBOOK* et *CREATE_NOTEPAD*. Pour connaître la quantité totale de mémoire non volatile déclarée, l'outil décompose le code à la recherche des morceaux de code effectuant ces appels et les paramètres associés. Le prototype est développé en python et utilise la fonction de recherche d'appels de fonctions de l'outil d'analyse de code radare2 [3]. Chaque instruction est codée sur 32-bits, ce qui facilite grandement cette décomposition. De plus, chaque fonction est construite sur la même structure : enregistrement des paramètres dans les registres R3 à R9, puis appel au service API. L'environnement de l'application étant très statique, ces services sont tous appelés avec des paramètres statiques sur notre cas d'étude.

4.1.2.2 Règles de calcul de l'utilisation des ressources

Une fois l'ensemble des appels à ces services et les paramètres utiles retrouvés, le montant total de ressources déclaré est calculé comme la somme du montant de chaque ressource déclarée, pour chacun de ces services, en utilisant les formules suivantes :

— **Mémoire RAM pour les communications inter-partition :** Deux types de services API ARINC 653 permettent d'effectuer des communications inter-partitions, les ports *SAMPLING* et les ports *QUEUING*. Les ports *SAMPLING* ne contiennent qu'un seul message, il suffit donc juste de connaître la taille maximale de ce message. Les ports *QUEUING* peuvent contenir plusieurs messages, il faut donc connaître la taille maximale de chaque message, et le nombre maximal de messages. Tous ces paramètres sont utilisés par les services *CREATE_SAMPLING_PORT* et *CREATE_QUEUING_PORT*.

```
CREATE_SAMPLING_PORT : MAX_MESSAGE_SIZE
CREATE_QUEUING_PORT : MAX_MESSAGE_SIZE * MAX_NB_MESSAGE
```

3. https ://rada.re/r/

— **Mémoire non volatile :** Deux types de services API permettent d'utiliser la mémoire non volatile, les *LOGBOOK* et les *NOTEPAD* (services non standardisés dans l'ARINC 653). Les *LOGBOOK* peuvent contenir plusieurs messages. De plus, leur recopie en mémoire non volatile n'est pas instantanée et nécessite une mémoire auxiliaire, où sont stockés les messages en attente. Pour connaître la quantité mémoire nécessaire lorsqu'un *LOGBOOK* est créé, il faut donc connaître le nombre maximal de messages enregistrés, le nombre maximal de messages pouvant être en transition, et la taille maximale des messages. Un *NOTEPAD* ne contient qu'un seul message. Il faut donc uniquement connaître la taille maximale de ce message. Tous ces paramètres sont utilisés par les services *CREATE_LOGBOOK* et *CREATE_NOTEPAD*.

```
CREATE_LOGBOOK : MAX_MESSAGE_SIZE * (MAX_NB_LOGGED_MESSAGE +
↪  MAX_NB_IN_PROGRESS_MESSAGE)
CREATE_NOTEPAD : MAX_MESSAGE_SIZE
```

— **Nombre de ports de communication :** Pour cette vérification, il suffit de compter le nombre d'utilisation des services *CREATE_SAMPLING_PORT* et *CREATE_QUEUING_PORT*.

```
CREATE_SAMPLING_PORT : 1
CREATE_QUEUING_PORT : 1
```

— **Services spécifiques :** Dans ce cas particulier, l'outil utilise une liste correspondant aux services spécifiques autorisés, et affiche l'ensemble des services utilisés n'étant pas dans la liste.

4.1.2.3 Limitations

Cet outil n'a été évalué que sur un seul cas d'étude, et n'a pas été évalué en cas de binaire malveillant. Il reste donc très limité, mais propose une réflexion sur la réutilisation des documents déjà disponibles lors du développement d'une application avionique afin de réaliser des vérifications statiques relatives à la sécurité, comme la vérification de l'usage correct des ressources par le binaire de l'application, au regard des ressources déclarées dans le contrat d'insertion. Cette étude n'a pas été poussée plus loin, le coeur de ces travaux étant plus focalisé sur la suite de l'approche (Définition du SDA et mise en place de la partie embarquée).

4.1.3 Moniteur de SDA

Le moniteur de SDA est utilisé pour surveiller l'exécution de l'application et collecter les données correspondantes. Ce moniteur est développé à l'aide de l'outil de *debug* GDB, qui permet un accès sans restriction aux informations du calculateur, tout en conservant une exécution temps-réel locale dans le calculateur. Concernant le niveau d'observation, le choix s'est porté sur les appels API ARINC 653 effectués

par l'application. A chaque appel effectué, le type de l'appel et son horodatage sont enregistrés. Deux versions de ce moniteur ont été développées au cours des travaux :

1. Utilisation du *debugger* : seul le *debugger* est utilisé pour enregistrer les informations choisies.
2. Insertion d'un code d'instrumentation : un code d'instrumentation est utilisé pour enregistrer en temps-réel les données d'observation dans une zone mémoire dédiée, et le *debugger* n'est utilisé que pour enregistrer ces données sur le contrôleur, après chaque slot d'exécution de l'application sous surveillance.

Dans les deux cas, les données sont enregistrées dans un fichier de log sur le contrôleur au format suivant :

```
12, 4003
14, 4451
1,  4808
7,  5133
8,  6107
...
```

La première colonne correspond à un identifiant relatif au type d'appel API utilisé. La deuxième colonne indique la valeur de l'horloge interne du RTOS, indiquée par le registre *TBL*.

4.1.3.1 Version 1 : Utilisation du *debugger*

La première version du moniteur consiste à placer des points d'arrêt dans le code de chaque appel API à surveiller. Dès que le point d'arrêt est atteint, l'information correspondante est écrite dans le fichier de log courant. Ces points d'arrêt sont placés à l'aide d'un script de *debug* ou script DBG. L'extrait de code suivant correspond à la partie du script DBG permettant de placer un point d'arrêt à l'appel *CREATE_PROCESS* et à enregistrer les informations voulues (type de l'appel, représenté ici par le numéro 1, et horodatage, contenu dans le registre *tbl*) :

```
b CREATE_PROCESS
commands
  silent
  printf "1,%i\n", $tbl
  c
end
```

Cette première version permet une grande souplesse dans le choix des informations à enregistrer. En particulier, cette version est intéressante pour tester rapidement différents choix de données à observer. Cependant, chaque point d'arrêt introduit un retard dû à l'échange d'information entre la cible et le contrôleur. Sur nos cas d'étude, l'observation de 20 secondes d'exécution de l'application prenait au final

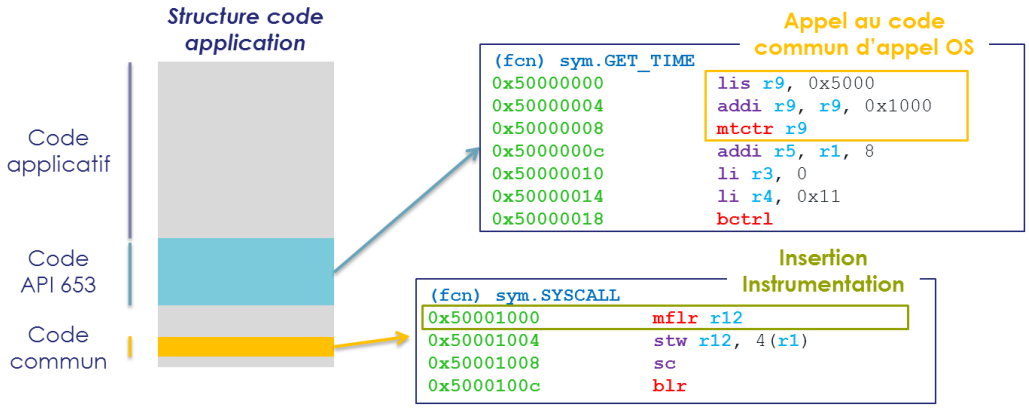


FIGURE 4.3 – Structure du code d'une application et insertion du code d'instrumentation

presque 30 minutes dû à l'accumulation de ces retards. La version suivante propose de réduire le nombre de points d'arrêt pour réduire cet écart entre la durée d'exécution sur le calculateur et la durée réelle de l'observation.

4.1.3.2 Version 2 : Insertion d'un code d'instrumentation

Dans la deuxième version, un code d'instrumentation est inséré directement dans le code de l'application. L'insertion de ce code est effectuée à l'initialisation du calculateur, via une commande GDB. Ce code est inséré directement dans la librairie d'appels API attachée à la partition surveillée. Plus précisément, chaque appel fait appel à un morceau de code commun, qui s'occupe ensuite d'appeler l'OS. La Figure 4.3 présente la structure du binaire de l'application, composé de son code applicatif (en gris), de la librairie API ARINC 653 (en bleu), et du code commun d'appel à l'OS (en jaune).

C'est dans cette fonction commune qu'est inséré le code d'instrumentation décrit par le Listing 4.1. Pour insérer ce morceau de code supplémentaire, l'instruction à l'adresse `0x50001000` est écrasée par un saut vers une zone de code non utilisée (ici, à l'adresse `0x10000000`). Le code d'instrumentation est placé à cette adresse (`0x10000000`), et se termine par l'exécution de l'instruction écrasée (ici, l'instruction *mflr r12*), puis un saut vers l'instruction normale suivante, ici à l'adresse `0x50001004`.

Le code additionnel d'instrumentation vérifie dans un premier temps si l'appel est dans la liste des appels exclus. C'est le cas des appels *LOCK_PREEMPTION*, *UNLOCK_PREEMPTION* et *TIMED_WAIT*, qui étaient appelés extrèmement souvent et engendraient donc une grande quantité de données. Ensuite, le code écrit dans une zone mémoire dédiée la valeur du registre R4, qui est utilisé comme identifiant de l'appel API, et la valeur du registre *TBL*, qui donne la valeur d'horloge interne du calculateur. Cette zone mémoire est composée de deux pointeurs

```
# Exclusion de certains appels API selon leur identifiant
0x10000000:         cmpwi   r4,3
0x10000004:         beq-    0x10000064
0x10000008:         cmpwi   r4,12
0x1000000c:         beq-    0x10000064
0x10000010:         cmpwi   r4,13
0x10000014:         beq-    0x10000064
0x10000018:         cmpwi   r4,15
0x1000001c:         beq-    0x10000064
# Sauvegarde des valeurs des registres R5 et R6
0x10000020:         stwu    r5,-4(r1)
0x10000024:         stwu    r6,-4(r1)
# Récupération du pointeur de logs courant, enregistré à l'adresse
↪  0x18000004
0x10000028:         li      r5,4
0x1000002c:         addis   r5,r5,6144
0x10000030:         lwz     r5,0(r5)
# Enregistrement de l'identifiant de l'appel, donné par le registre R4
0x10000034:         stw     r4,0(r5)
0x10000038:         addi    r5,r5,4
# Enregistrement de l'horodatage
0x1000003c:         mftb    r6
0x10000040:         stw     r6,0(r5)
0x10000044:         addi    r5,r5,4
# Mise à jour du pointeur de logs
0x10000048:         li      r6,4
0x1000004c:         addis   r6,r6,6144
0x10000050:         stw     r5,0(r6)
# Restauration des registres R5 et R6
0x10000054:         lwz     r6,0(r1)
0x10000058:         addi    r1,r1,4
0x1000005c:         lwz     r5,0(r1)
0x10000060:         addi    r1,r1,4
# Exécution de l'instruction écrasée par le saut vers le code
↪  d'instrumentation
0x10000064:         mflr    r12
0x10000068:         b       0x50001004 <SYSCALL+4>
```

Listing 4.1 – Code additionnel d'instrumentation (ASM)

représentant la première et la dernière case remplie de la fenêtre de logs, et d'une fenêtre de logs.

L'enregistrement des logs dans un fichier sur le contrôleur est effectué à l'aide d'un script DBG. Un point d'arrêt est placé au moment où l'OS reprend la main après le slot d'exécution de l'application surveillée. Le script parcourt ensuite la fenêtre de logs, d'après les valeurs des pointeurs de début et de fin, pour enregistrer chaque log dans le fichier de données sur le contrôleur. Il réinitialise ensuite la fenêtre de logs pour le prochain slot d'exécution.

4.1.4 Prétraitement des données

Le prétraiement des données consiste à agréger les données pour qu'elles soient compréhensibles par les outils de modélisation et de vérification du SDA. Dans cette première implémentation, les données sont agrégées sous la forme de séquences d'appels API, accompagnées de leur durée. D'autres versions sont proposées et étudiées de façon plus approfondie dans le Chapitre 5.

Les données sont agrégées selon une taille de séquence donnée. Si cette taille est de 4 appels API, les données seront agrégées sous la forme [ID1, ID2, ID3, ID4, durée]. La durée de la séquence est calculée comme la différence entre l'horodatage du premier et du dernier appel de la séquence. Une fenêtre glissante est utilisée pour construire les données prétraitées. Si l'on considère les données d'exemple de la Section 4.1.3, le résultat du prétraitement, avec des séquences de taille 4, sera le suivant :

```
[12, 14, 1, 7, 1130]
[14, 1, 7, 8, 1656]
```

Ces données peuvent également être représentées en deux dimensions en concaténant les identifiants des appels API. Cette concaténation permet de donner un identifiant unique à une séquence :

```
[12140107, 1130]
[14010708, 1656]
```

Les données subissent ensuite deux transformations supplémentaires. D'abord, l'identifiant de séquence est transformé selon un encodage *one-hot*. Cette forme d'encodage consiste à coder une valeur sur un vecteur de n bits, tel que la somme des éléments du vecteur soit égale à 1. C'est-à-dire que seule une case du vecteur vaut 1, tandis que toutes les autres prennent la valeur 0. Dans notre cas, si M différentes séquences **normales** sont observées pour une application donnée (parmi toutes les N^t séquences possibles, N étant le nombre d'appels API différents et t étant la taille de la séquence), l'identifiant d'une séquence sera codé sur un vecteur de M valeurs. Toutes les valeurs du vecteur seront égales à 0, excepté la valeur à la position i, i correspondant à la position donnée pour un identifiant de séquence particulier. Dans

notre cas, si la séquence courante n'existe pas (parmi les M séquences **normales**), l'ensemble des valeurs du vecteur prendront la valeur 0.

Ce type d'encodage permet de représenter des données qualitatives (qui représentent des catégories dont la valeur numérique n'a pas de sens en elle-même), qui ne sont donc pas hiérarchisées. Dans ce sens, associer un nombre à chaque séquence légitime introduirait des liens entre séquences qui n'ont pas de sens dans notre contexte (par exemple une séquence 1-1 n'est pas plus proche d'une séquence 1-2 que d'une séquence 48-37). Ce type d'encodage est donc nécessaire pour représenter correctement ces données, et pouvoir y appliquer des méthodes d'apprentissage automatique, notamment OCSVM.

En reprenant l'exemple précédent, si l'on considère que seules les trois séquences [07,08,12,14], [12,14,01,07], et [14,01,07,08] sont normales, le résultat de ce prétraitement sera le suivant :

```
[0, 1, 0, 1130]
[0, 0, 1, 1656]
```

La dernière transformation consiste à rapporter les valeurs correspondant à la durée sur une échelle entre 0.0 et 1.0, en fonction des valeurs minimale ($durée_{min}$) et maximale ($durée_{max}$) rencontrées. Cette transformation est appelée *scaling* en anglais. La formule de calcul est la suivante :

$$Scaling : durée_{finale} = \frac{durée_{initiale} - durée_{min}}{durée_{max} - durée_{min}} \tag{4.1}$$

Sur les données d'exemple, si les valeurs de durée minimale et maximale sont respectivement 500 et 2000, on obtient les données finales suivantes :

```
[0, 1, 0, 0.42]
[0, 0, 1, 0.77]
```

Une fois l'ensemble des transformations réalisées, l'outil enregistre ces données prétraitées dans un nouveau fichier. Chaque fichier de données brutes obtenu par le moniteur de SDA est donc transformé en un nouveau fichier de données prétraitées correspondant.

4.1.5 Modélisation du SDA

La modélisation du SDA se base sur des techniques d'apprentissage semi-supervisé pour définir le modèle de comportement normal de l'application surveillée, à partir des données prétraitées par l'outil précédent. L'outil de modélisation se base uniquement sur des données normales pour créer ce modèle de comportement, utilisé ici comme SDA. Plusieurs techniques existent, comme souligné dans le Chapitre 2. Dans ces travaux, cette partie de modélisation a été implémentée en python, à l'aide de la librairie *scikit-learn*[4]. Cette librairie implémente un grand

4. https ://scikit-learn.org

nombre d'algorithmes d'apprentissage automatique, notamment celui de *One-Class Support Vector Machine* (OCSVM), qui a été utilisé dans cette première implémentation. Cet algorithme, décrit dans [Scholkopf 2001], a montré son efficacité avec succès sur différents problèmes de modélisation de comportement normal, tels que décrits dans le Chapite 2. Son utilisation pour développer un NIDS avionique dans [Gil Casals 2014], sa facilité de configuration, et ses performances en terme de rapidité de détection nous ont poussé à l'utiliser dans le cadre d'une première implémentation de notre HIDS. En particulier, la rapidité d'exécution de la partie de détection est un critère très important dans notre approche, puisque cette partie de détection sera effectuée en vol. Au contraire, optimiser le temps nécessaire à la phase d'apprentissage est moins important dans ce contexte, puisque cette phase est exécutée au sol, avec des ressources plus importantes à disposition. Une réflexion et des études plus approfondies relatives au choix de l'algorithme d'apprentissage sont proposées dans le Chapitre 5.

Le principe de cet algorithme est de sélectionner, parmi les données d'apprentissage, celles qui représentent le mieux l'ensemble des données d'apprentissage. Ces vecteurs sont appelés les **vecteurs support**. Un poids est attaché à chaque vecteur support, et permet d'indiquer l'importance de chaque vecteur dans la constitution générale du modèle. Trois principaux paramètres doivent être définis pour cette phase d'apprentissage :

— $kernel \in \{Linéaire, RBF, Polynomial\}$: Définit l'aspect général du modèle OCSVM.

— $nu \in]0; 1]$: Donne le taux de faux positifs acceptable.

— $gamma \in N$: Précise à quel point le modèle doit être proche des points d'entraînement.

La librairie *scikit-learn* facilite grandement l'implémentation de cette phase d'apprentissage, en proposant directement la fonction *fit*, qui permet de sélectionner les vecteurs support et leur poids directement à partir d'un ensemble de vecteurs représentant le comportement normal de l'application. Pour l'algorithme d'apprentissage de l'OCSVM, l'étape de prétraitement est très importante, notamment la phase de transformation en encodage *one-hot* et le *scaling*.

4.1.6 Injection d'attaque

Le principe de l'injection d'attaque développé dans ce prototype est d'effectuer des modifications de code directement dans la mémoire RAM contenant le code de l'application ciblée. Cette modification permet d'émuler un code malveillant exécuté par l'application surveillée, et donc des comportements anormaux, qui serviront à tester l'efficacité de détection de l'HIDS. Cette modification est effectuée lors de l'initialisation du calculateur, en utilisant un script GDB. Aucune autre interaction n'est effectuée par l'outil d'injection pendant la durée de l'expérimentation. Le script GDB remplace ou ajoute des instructions selon une stratégie prédéfinie. L'implémentation de cet outil fait l'objet d'une section à part entière, la Section 4.2.

4.1.7 Vérification du SDA

Pour finir, l'étape de vérification du SDA a également été développée en python, à l'aide de la librairie *scikit-learn*. Cette librairie propose deux fonctions permettant de déterminer si une donnée se situe dans le modèle OCSVM ou non :

— *predict* : Cette fonction renvoie la valeur 1 si la donnée est considérée comme appartenant au modèle, et la valeur -1 si elle est considérée en-dehors.

— *decision_function* : Cette fonction permet de connaître la distance entre la donnée évaluée et le modèle. Une distance négative signifie que la donnée n'appartient pas au modèle. Cependant, cette fonction permet d'ajouter artificiellement une marge autour du modèle, pour accepter les points très proches de la limite comme étant normaux. Cela permet une plus grande flexibilité dans l'exploitation du modèle, notamment pour éviter de générer un trop grand nombre de faux positifs. Ce paramètre sera appelé *tolérance* dans la suite de ce manuscrit.

Le principe de la vérification de l'OCSVM est de calculer une distance entre la donnée courante et l'ensemble des vecteurs supports, associés à leur poids respectif. Si la valeur finale de distance est négative, la donnée est considérée comme hors du modèle. La phase de vérification est implémentée telle qu'un fichier de données prétraitées est considéré comme contenant une attaque si au moins P anomalies sont relevées dans un intervalle de temps donné, ici 0.5 secondes (ce qui correspond à 10 cycles d'exécution de l'application cible). Ce dernier paramètre P est appelé ici le *seuil*.

Enfin, si le SDA considéré n'est pas suffisamment efficace, le prototype propose des outils permettant d'étudier la distribution des durées de chaque séquence, la distribution des anomalies par type de séquence ou encore les distances des anomalies relevées par rapport au modèle. Ces observations permettent de comprendre comment faire évoluer les paramètres de l'OCSVM lors de l'apprentissage, ou la tolérance ou le seuil à utiliser, pour améliorer les résultats.

4.2 Implémentation de l'outil d'injection

L'outil d'injection d'attaque a été implémenté sous la forme de scripts GDB et python. Il est utilisé pendant la phase d'intégration, afin de générer des comportements anormaux. Il est implémenté sur le contrôleur afin d'interagir avec la cible. Plus précisément, ce sont les scripts GDB qui permettent d'interagir avec la cible (démarrage de l'application, injection du code malveillant, extraction des logs, redémarrage et restauration du calculateur), tandis que les scripts python sont utilisés pour implémenter les scanners et le générateur. Des scripts python sont également utilisés pour générer des scripts GDB spécifiques au cours de la campagne d'injection, qui vont injecter l'attaque courante et définir le nom du fichier de logs à utiliser. L'injection est réalisée à l'initialisation du calculateur afin de simuler

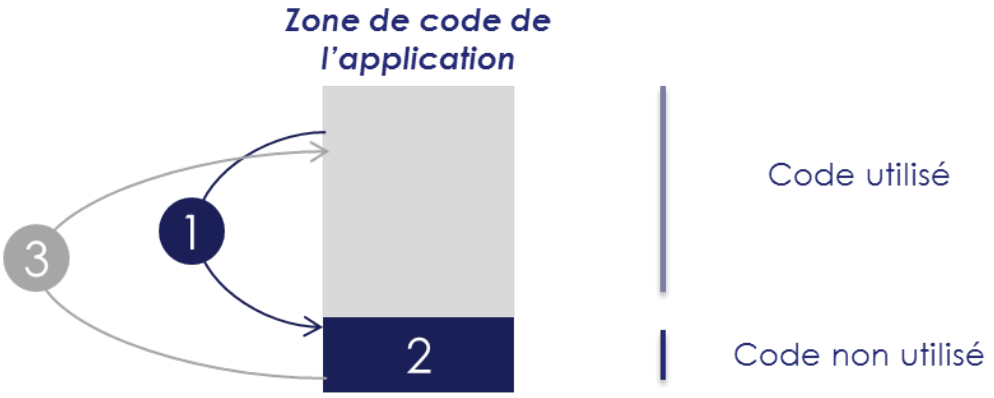


FIGURE 4.4 – Principe de l'ajout de code

un FLS malveillant. Cette injection est réalisée par une modification directe de la mémoire RAM via l'utilisation de GDB.

Cette section présente les spécificités implémentées dans le prototype d'outil d'injection d'attaques réalisé. Plus précisément, cette section présente la façon dont sont implémentés 1) l'ajout de code, 2) le contrôle de l'activation de la charge malveillante, 3) les opérateurs d'attaque, et 4) la gestion de la campagne d'injection.

4.2.1 Ajout de code

L'insertion de code supplémentaire est réalisée à l'aide d'une zone de code non utilisée, déjà présente à l'intérieur de l'application. En effet, si l'on considère un scénario donné, l'ensemble du code de l'application ne sera pas exécuté. Les zones de code non utilisées sont donc reprises dans ce prototype afin de simuler l'ajout de code malveillant. La Figure 4.4 présente le principe utilisé pour cet ajout de code. A l'emplacement où l'injection de code supplémentaire est réalisée, l'instruction originale est remplacée par un saut vers une zone de code non utilisée (1). Le code supplémentaire est stocké à cet emplacement, suivi de l'instruction qui a été supprimée pour insérer le saut (2). La dernière instruction insérée dans la zone de code non utilisée est également un saut, qui permet de continuer l'exécution normale de l'application (3).

4.2.2 Contrôle de l'activation de la charge malveillante

Deux types de contrôles ont été implémentés pour définir le moment d'activation de la charge malveillante. Ces deux contrôles se basent sur la valeur d'un compteur qui est incrémenté à chaque exécution du code injecté. Le premier contrôle est un contrôle temporel, et consiste à définir un intervalle pendant lequel la charge malveillante sera exécutée. Le deuxième contrôle consiste à définir une fréquence à laquelle la charge malveillante est injectée (contrôle fréquentiel).

Le pseudo-code suivant correspond au code ajouté pour effectuer ces deux contrôles simultanément :

```
compteur++;
# Vérifier l'intervalle du compteur
if début < compteur < fin :
  # Vérifier la fréquence
  if compteur % N == 0 :
    Exécuter la charge malveillante;
```

Pour les opérateurs implémentant ces deux contrôles, trois paramètres doivent être définis :

— *début*, utilisé pour le contrôle temporel, correspond au nombre d'exécutions du code malveillant à réaliser avant d'activer la charge malveillante,

— *fin*, utilisé pour le contrôle temporel, correspond au nombre d'exécutions du code malveillant à partir de laquelle la charge malveillante ne doit plus être activée, et

— N, utilisé pour le contrôle fréquentiel, correspond à la fréquence à laquelle la charge malveillante doit être activée (1 fois sur N).

4.2.3 Opérateurs d'attaque implémentés

Dans ce prototype, 9 opérateurs d'attaque ont été implémentés. Étant donné le temps nécessaire à l'implémentation d'un opérateur, seul un sous-ensemble des types d'opérateur tels que définis dans la Section 3.3.3 a été implémenté, de telle sorte que chaque stratégie (*CrashMe*, *Substitution d'instruction(s) bien formée(s)*, et *Motif d'attaque*) et chaque spécificité définie précédemment (ajout de code et contrôle temporel/fréquentiel de l'activation de la charge malveillante) soient utilisées. La Table 4.2 recense les différents opérateurs implémentés avec la stratégie et les spécificités associées.

Plus précisément, les opérateurs ont été implémentés sous la forme suivante :

— MIA : Remplace une instruction par une valeur aléatoire

— MI : Remplace n instructions consécutives par n instructions consécutives du dictionnaire (pour cette implémentation, $n = 4$ a été choisi arbitrairement)

— MS_1 : Modifie la condition de l'instruction de saut

— MS_2 : Modifie la destination de l'instruction de saut

— SI : Remplace une instruction par un *NOP*

— AI : Duplique l'instruction

— MR : Modifie la valeur d'un registre par une valeur aléatoire

— MV : Modifie la valeur d'une case mémoire par une valeur aléatoire

— SF : Exécute une instruction *RET* de retour de fonction au point d'entrée de la fonction

TABLE 4.2 – Caractéristiques des opérateurs d'attaque implémentés

Opérateur	Stratégie	Ajout de code	Contrôle d'activation temporel	Contrôle d'activation fréquentiel
MIA	*CrashMe*			
MI	*Substitution d'instruction(s) bien formée(s)*			
MS_1/MS_2/SI	*Motif d'attaque*			
AI	*Motif d'attaque*	×		
MR/MV	*Motif d'attaque*	×	×	
SF	*Motif d'attaque*	×	×	×

4.2.4 Gestion de la campagne d'injection

La campagne d'injection est gérée automatiquement à l'aide d'un script GDB, tel que l'exemple donné dans le Listing 4.2.

Ce script correspond à la première version d'instrumentation proposée dans la Section 4.1.3, basée sur des points d'arrêt GDB. L'utilisation de la 2ème version d'instrumentation, basée sur l'insertion d'un code d'instrumentation, permet de simplifier grandement ce script de campagne d'injection d'attaque, puisqu'il n'y a plus qu'un point d'arrêt GDB utilisé pour l'observation.

Ce script est généré automatiquement à l'aide d'un script python, en fonction de la configuration de l'outil. Pour le script présenté ici, les éléments de configuration utilisés sont le point d'entrée de l'application, et les points d'observation pour le moniteur de SDA (le service GET_TIME avec l'ID n°1 et le service SEND_QUEUING_MESSAGE avec l'ID n°2). La procédure de restauration et redémarrage du calculateur est enregistrée dans une fonction GDB appelée *restore_and_restart_module*.

Ce script se décompose en trois phases, détaillées par la suite : 1) la déclaration des points d'arrêt, 2) le lancement de la boucle sur les attaques de la liste d'attaques, et 3) la gestion des points d'arrêts pendant une exécution. La suite de la section détaille également la fonction *retrieve_and_inject_attack*, utilisée pour injecter une attaque, puis donne un exemple de fichier de log généré à l'aide de ce script.

4.2.4.1 Déclaration des points d'arrêt (lignes 1-7)

Dans un premier temps, le script déclare les points d'arrêt nécessaires pour injecter l'attaque et enregistrer les informations d'observation de l'application. Dans cet exemple, trois points d'arrêt sont déclarés : le point d'entrée de l'application, qui indique le moment où l'attaque sera injectée (*APP_ENTRY_POINT*), et deux points d'observation pour le moniteur de SDA (les deux services API ARINC 653 *GET_TIME* et *SEND_QUEUING_MESSAGE*). Ces informations sont déclarées

```
# 1. Définition des points d'arrêt pour l'observation
# APP_ENTRY_POINT
b *0x30000000
# GET_TIME
b *0x10000000
# SEND_QUEUING_MESSAGE
b *0x20000000
# 2. Initialisation
# VARIABLES LOCALES POUR LA CAMPAGNE D'INJECTION
set $MAX_COUNTER = 4
set $current_counter = -1
# CONTINUE JUSQU'À ATTEINDRE UN PREMIER POINT D'ARRÊT
continue
# BOUCLE POUR LANCER LES ATTAQUES AUTOMATIQUEMENT
while ($current_counter < $MAX_COUNTER)
  # 3. Gestion des points d'arrêt
  if $pc==0x10000000||$pc==0x20000000||$pc==0x30000000||$pc==$addr_attack
    # RESTAURER ET REDÉMARRER LE CALCULATEUR
    if  $tbl >= $max_duration_tbl
      set logging off
      clear *$addr_attack
      restore_and_restart_module
    end
    # APP_ENTRY_POINT (INJECTION DE L'ATTAQUE)
    if $pc == 0x30000000
      retrieve_and_inject_attack
    end
    # GET_TIME
    if $pc == 0x10000000
      printf "1,0x%x\n",$tbl
    end
    # SEND_QUEUING_MESSAGE
    if $pc == 0x20000000
      printf "2,0x%x\n",$tbl
    end
    # EXÉCUTION DE L'ATTAQUE
    if $pc == $addr_attack
      printf "ATTACK_LAUNCHED,0x%x\n",$tbl
    end
  # CRASH DE L'APPLICATION
  else
    # RESTAURER ET REDÉMARRER LE CALCULATEUR
    set logging off
    clear *$addr_attack
    restore_and_restart_module
  end
  continue
end
```

Listing 4.2 – Script de gestion de la campagne d'injection (Script GDB)

dans un fichier de configuration dédié.

4.2.4.2 Lancement de la boucle sur la liste d'attaques (lignes 8-15)

Le nombre d'attaques à lancer est récupéré directement depuis la liste d'attaques et stocké dans une variable $\$MAX\_COUNTER$. Le script déclare également un compteur d'attaque courant, et exécute une boucle sur l'ensemble des attaques de la liste à l'aide de ces deux variables. L'instruction *continue* de la ligne 13 permet au script de continuer l'exécution de la cible, jusqu'à atteindre un premier point d'arrêt.

4.2.4.3 Gestion des points d'arrêt (lignes 16-48)

Lorsqu'un point d'arrêt est atteint, le script reprend la main. C'est donc la partie de gestion des points d'arrêt qui va s'exécuter. Dans un premier temps, le script vérifie que le point d'arrêt atteint est bien un point prévu (ligne 17). Si ce n'est pas le cas, c'est que l'application a rencontré une exception et ne peut plus continuer son exécution correctement. Le script passe donc à l'attaque suivante (lignes 40-46).

Si c'est un point d'arrêt prévu, le script vérifie d'abord la durée de l'expérimentation (lignes 18-23). Si la durée maximale définie dans la configuration est atteinte, le script lance la prochaine expérimentation (la prochaine attaque sur la liste).

Sinon, l'expérimentation courante continue, et le script effectue un traitement en fonction du point d'arrêt atteint. Celui-ci est reconnu à l'aide du pointeur d'instruction courante enregistré dans le registre *$pc*. Si l'on est au point d'entrée de l'application (lignes 24-27), l'attaque est injectée à l'aide de la fonction *retrieve_and_inject_attack*. Si c'est un point d'observation (lignes 28-35), l'information correspondante est enregistrée avec la valeur du registre *$tbl*, qui correspond à une valeur d'horloge interne au calculateur. Enfin, un dernier point d'observation (lignes 36-39) permet d'indiquer, dans le fichier de logs, que le code injecté est exécuté.

Enfin, le script redonne la main à la cible avec l'instruction *continue* (ligne 47).

4.2.4.4 Injection de l'attaque (fonction *retrieve_and_inject_attack*, ligne 26)

La fonction d'injection de l'attaque génère d'abord un script GDB correspondant à l'attaque courante, en fonction du compteur $\$current\_counter$, puis l'exécute. Le code suivant est un exemple d'un tel script GDB :

```
1 set $current_counter = $current_counter + 1
2 set $addr_attack = 0x40000000
3 b* $addr_attack
4 AI 0x40000000
5 set logging file data_AI-0x40000000.txt
6 set logging on
```

La première ligne met à jour le compteur d'attaques. La deuxième ligne indique l'adresse à laquelle l'attaque est injectée, tandis que la troisième place un point d'arrêt à cet emplacement. Cette adresse est utilisée pour enregistrer un log lorsque le code d'attaque est atteint. La mutation de l'application est réalisée par la ligne 4 (sur cet exemple, l'opérateur AI est utilisé avec un seul paramètre). Enfin, les deux dernières lignes permettent de configurer le fichier de logs correspondant à cette attaque.

4.2.4.5 Fichiers de log générés

Un fichier de logs est généré pour chaque expérimentation réalisée pendant la campagne d'injection. Chaque ligne du fichier correspond à un enregistrement, soit relevé par un point d'observation (ici, ID n°1 et ID n°2), soit par l'exécution du code malveillant (tag *ATTACK_LAUNCHED*). Dans tous les cas, une seule attaque est injectée par expérimentation. S'il y a plusieurs fois le tag *ATTACK_LAUNCHED* dans un même fichier de logs, cela signifie que le code malveillant injecté au démarrage de l'application a été exécuté plusieurs fois pendant une même expérimentation. L'exemple suivant est un extrait de fichier de logs généré suivant le script de campagne décrit dans cette section :

```
2,0x14b444d2
1,0x14b449da
1,0x14b44fe6
2,0x14b4535d
ATTACK_LAUNCHED,0x14b45563
1,0x14b46e06
2,0x14b470cb
2,0x14b47441
```

4.3 Expérimentations

Cette section décrit deux expérimentations réalisées à l'aide du prototype décrit dans ce chapitre. La première expérimentation vise à démontrer l'intérêt de l'outil pour calibrer un HIDS avionique de manière efficace, sans connaître des exemples d'attaques réelles. La deuxième expérimentation cherche à évaluer plus précisément l'efficacité de cet HIDS, par une étude de sensibilité en fonction des opérateurs d'attaque utilisés. Au préalable, la première partie de cette section présente le protocole d'expérimentation utilisé.

4.3.1 Processus d'évaluation de l'HIDS

Le processus d'évaluation de l'HIDS proposé suit les concepts de l'apprentissage automatique avec trois phases principales : l'entraînement, le test, et l'évaluation. Un jeu de données labellisé est associé à chaque phase, c'est-à-dire, un ou plusieurs

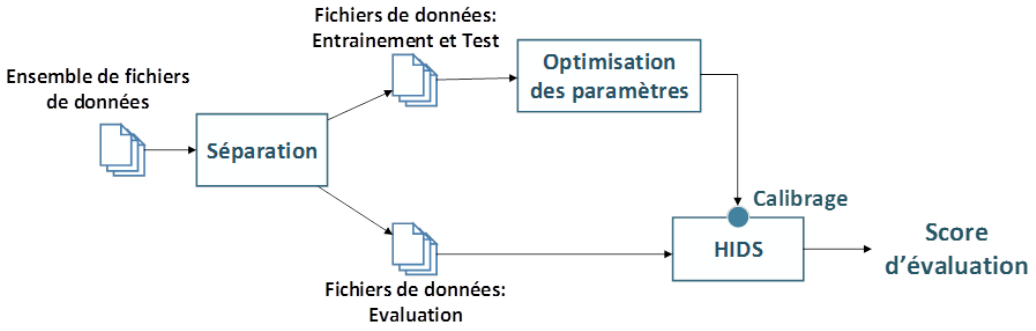


FIGURE 4.5 – Utilisation des fichiers de données pour les phases d'entraînement, de test, et d'évaluation de l'HIDS

fichier(s) de données (ici, des fichiers de logs), identifiés comme résultant d'une exécution normale ou d'une exécution avec injection d'attaque. La Figure 4.5 propose un schéma simplifié de l'utilisation de ces jeux de données dans notre contexte. L'ensemble des fichiers de log est séparé en deux parties. La première partie est utilisée pour l'entraînement et le test, et la deuxième partie est utilisée pour l'évaluation. La phase d'entraînement permet de créer un modèle, tandis que la phase de test permet de donner un score à ce modèle. Ces deux phases sont exécutées plusieurs fois afin d'optimiser le score donné lors de la phase de test (**Optimisation des paramètres**). Dans notre cas, cette phase permet d'ajuster les paramètres de l'HIDS (**Calibrage**) pour qu'il soit le plus efficace possible sur les données de test. Enfin, la phase d'évaluation n'est exécutée qu'une seule fois. Lors de cette phase, chaque fichier de données d'évaluation est évalué par l'HIDS précédemment calibré et donne un score de détection unique, sur les données d'évaluation (**Score d'évaluation**). Pour ces expérimentations, le score de détection choisi est le *rappel* si la *précision* est de 100%. Si ce n'est pas le cas, le score est de 0. Ce score permet de donner la priorité à l'absence de faux positifs, puis à optimiser le nombre d'attaques détectées.

Plus précisément, les phases d'entraînement et de test sont réalisées selon le schéma de la Figure 4.6. Parmi les fichiers de log, un seul est utilisé pour la phase d'entraînement (ici, **f1**). Ce fichier doit être labellisé comme étant normal. Éventuellement, une phase d'initialisation (**1.1**) peut être nécessaire, par exemple pour connaître les différentes séquences d'appels API légitimement utilisées par l'application (qui apparaissent donc dans les données d'entraînement). Ensuite, un prétraitement (**1.2**) est appliqué à chaque fichier de données, individuellement. Chaque fichier de données brutes engendre donc un fichier de données pré-traitées. Le fichier de données pré-traité correspondant au fichier choisi pour l'entraînement (ici, **dp1**) est utilisé pendant la phase d'entraînement (**2**) pour apprendre un modèle. Ce modèle est utilisé pendant la phase de test (**3**) pour évaluer si chaque fichier de données pré-traité restant (ici, **dp2** et **dp3**) contient une attaque ou non. Le score de détection final est calculé en fonction des labels prédits, par rapport aux

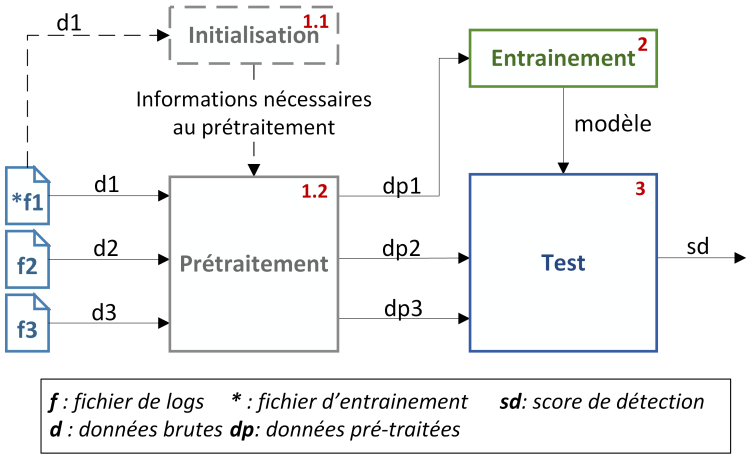


FIGURE 4.6 – Processus d'entraînement et de test

véritables labels des fichiers de données.

Ces phases sont exécutées de multiples fois, afin de sélectionner le meilleur score obtenu selon différents paramétrages. En particulier, le choix du fichier d'entraînement, les paramètres de pré-traitement, les paramètres d'apprentissage du modèle, et les paramètres de détection constituent l'ensemble des paramètres utilisés ici. Chaque paramètre possède un intervalle de valeur prédéfini. Le meilleur score est sélectionné après un calcul exhaustif des scores pour l'ensemble des combinaisons de paramètres prédéfinies. Cette technique, appelée *grid-search* en anglais, est très courante dans le domaine de l'apprentissage machine [5].

4.3.2 Expérimentation 1 : pertinence de l'outil

Pour cette première expérimentation, la cible exécute l'application d'affichage IHM-DA, qui renseigne le pilote sur l'état de son aéronef. C'est cette application IHM-DA qui est observée dans cette expérimentation. L'outil d'injection d'attaque a été utilisé pour générer les fichiers de données suivants, qui constituent l'ensemble des fichiers de données utilisés ici :

— 49 fichiers de données d'attaque générés avec une campagne d'injection aléatoire, d'une durée de 10 à 30 secondes,

— 5 fichiers de données normaux, d'une durée de 10, 20, 20, 20 et 30 secondes,

— 51 fichiers de données d'attaque générés à partir de 9 scénarios d'attaques précis, implémentés à l'aide une liste d'attaques produite manuellement, d'une durée de 10 à 30 secondes.

A titre informatif, la Table 4.3 donne quelques caractéristiques de l'application IHM-DA observées dans les fichiers de données normaux, relatives à son utilisation normale des appels API.

5. https ://scikit-learn.org/stable/modules/grid_search.html

TABLE 4.3 – Caractéristiques de l'application IHM-DA observées à partir des fichiers de données normaux

Nombre d'appels API effectués par slot d'exécution	83
Proportion d'appels API relatifs aux communications	33.7%
Nombre d'appels API différents	23
Nombre de séquences d'appels API différentes	tous (communications)
taille des séquences = 2	31 (9)
taille des séquences = 3	42 (13)
taille des séquences = 4	52 (17)
taille des séquences = 5	62 (21)

TABLE 4.4 – Paramètres de l'HIDS utilisés pour optimiser les résultats de détection

Phase	Paramètre	Valeurs évaluées	Valeur retenue
-	Choix du fichier d'entraînement	[0, 1, 2, 3, 4]	[4]
Prétraitement	Taille des séquences	[2, 3, 4, 5]	[3]
Apprentissage (OCSVM)	*kernel*	[RBF]	[RBF]
Apprentissage (OCSVM)	*nu*	[0.01]	[0.01]
Apprentissage (OCSVM)	*gamma*	[1000]	[1000]
Test	Tolérance	range(0.0,1.0,0.1)	[0.7]
Test	Seuil	range(0,80,1)	[2]

Les phases d'entraînement et de test ont été effectuées avec les 49 fichiers de données d'attaques aléatoires et les 5 fichiers de données normaux. En particulier, les différents paramètres et les valeurs utilisées pour chacun sont détaillés dans la Table 4.4 (Valeurs évaluées).

Tout d'abord, il est nécessaire de sélectionner le fichier qui sera utilisé pour l'entraînement. Pour la phase de prétraitement des données, où celles-ci sont agrégées sous la forme de séquences d'appels API représentées sous la forme *one-hot encoding* suivi de l'application d'un *scaling*, le paramètre considéré est la *taille des séquences*. Pour la phase d'entraînement, les paramètres sont directement liés à la technique d'apprentissage utilisée, ici l'OCSVM. Enfin, pour la phase de test, les paramètres considérés ici sont la *tolérance* (application d'une marge autour du modèle) et le *seuil* (nombre d'anomalies acceptables dans une fenêtre de temps donnée). A l'issu de la phase d'optimisation des paramètres de l'HIDS, le meilleur score de détection obtenu a été de 94.23%. Ce score correspond au meilleur *rappel* obtenu, pour 100% de *précision*. La Table 4.4 présente également les paramètres finaux utilisés pour calibrer l'HIDS (Valeur retenue).

TABLE 4.5 – Scénarios d'attaque ciblés exécutés sur l'application IHM-DA

ID	Impact	Description
1	Mécanismes de tolérance aux fautes	Suppression des appels au service de gestion d'erreurs applicatives
2	Intégrité	Modification de la valeur d'une donnée affichée
3	Intégrité	Modification continue des noms de menus affichés
4	Intégrité	Modification partielle des noms de menus affichés
5	Disponibilité	Suppression de tous les appels au service de libération de la préemption d'exécution
6	Disponibilité	Suppression de certains appels au service de libération de la préemption d'exécution
7	Disponibilité	Clignotement constant de l'affichage
8	Disponibilité	Clignotement partiel de l'affichage (à partir de 2 secondes et pendant 3 secondes)
9	Disponibilité	Destruction de la structure de l'affichage

Les 51 fichiers de données d'attaque générés à partir de scénarios précis ont ensuite été utilisés pour la phase d'évaluation. La Table 4.5 détaille les 9 scénarios utilisés sur l'application IHM-DA. Ces scénarios visent la disponibilité, l'intégrité, ou les mécanismes de tolérance aux fautes de l'application.

Avec le calibrage précédent, l'HIDS a été capable de détecter l'ensemble de ces fichiers de données comme contenant des attaques, c'est-à-dire que l'on a obtenu un score d'évaluation de 100%. Ce résultat permet de montrer qu'en utilisant l'outil de façon aléatoire, nous avons été capables de calibrer un HIDS de façon à détecter des attaques spécifiquement conçues pour avoir un impact réel. Cela signifie que l'approche, combinée avec l'outil, est capable de définir un modèle de comportement de l'application suffisamment précis pour détecter des attaques potentielles, même si aujourd'hui, aucun exemple réel d'attaque n'est disponible.

4.3.3 Expérimentation 2 : efficacité de détection de l'HIDS

Cette deuxième expérimentation cherche à caractériser plus précisément les résultats de détection, par rapport aux opérateurs d'attaque utilisés. Elle a été menée sur la deuxième application d'affichage cockpit, appelée IHM-DV. Cette application donne des informations de vol au pilote, comme sa vitesse ou son cap. Les fichiers suivants ont été générés à l'aide de l'outil d'injection :

— 20 exemples d'attaques pour chacun des 9 opérateurs implémentés, soit 180 fichiers de données d'attaque, d'une durée de 20 secondes chacun,

— 51 fichiers de données normaux, d'une durée de 10 à 30 secondes.

Pour cette expérimentation, chaque opérateur d'attaque a donc été utilisé 20 fois avec des paramètres choisis aléatoirement. A titre informatif, la Table 4.6 donne

TABLE 4.6 – Caractéristiques de l'application IHM-DV observées à partir des fichiers de données normaux

Nombre d'appels API effectués par slot d'exécution	173
Proportion d'appels API relatifs aux communications	82.1%
Nombre d'appels API différents	30
Nombre de séquences d'appels API différentes	tous (communications)
taille des séquences = 2	43 (25)
taille des séquences = 3	90 (56)
taille des séquences = 4	124 (88)
taille des séquences = 5	145 (106)

TABLE 4.7 – Scores d'évaluation obtenus pour les cinq expérimentations réalisées

N°	Durée du fichier d'entraînement	Taille des séquences	Précision (Test)	Rappel (Test)	Précision (Évaluation)	Rappel (Évaluation)
1	30s	3	100%	100%	100%	84.57%
2	25s	3	100%	100%	99.30%	87.04%
3	30s	3	100%	100%	100%	85.80%
4	30s	3	100%	100%	100%	87.65%
5	30s	3	100%	94.44%	100%	84.57%

quelques caractéristiques de l'application IHM-DV observées dans les fichiers de données normaux, relatives à son utilisation normale des appels API.

Pour les deux phases d'entraînement et de test, un total de 23 fichiers sont utilisés : 5 fichiers normaux sont sélectionnés aléatoirement parmi les 51 fichiers de données normaux, et 10% des fichiers d'attaque sont également utilisés (18 fichiers). De la même manière que la première expérimentation, les paramètres de l'HIDS sont optimisés sur ces 23 fichiers (1 fichier normal utilisé pour la phase d'entraînement, et les 22 fichiers restants utilisés pour la phase de test).

L'évaluation est ensuite réalisée sur l'ensemble des fichiers restants, soit 46 fichiers normaux et 162 contenant une attaque. Cette expérimentation a été menée 5 fois, avec à chaque fois une séparation différente entre les fichiers utilisés pour les phases d'entraînement et de test, et les fichiers utilisés pour la phase d'évaluation. La Table 4.7 résume les résultats obtenus pour chacune de ces 5 expérimentations, en précisant la durée du fichier d'entraînement sélectionné pour l'évaluation, la taille de séquence sélectionnée pour l'évaluation, et les valeurs de précision et rappel obtenues lors des phases de test et d'évaluation.

Pour chaque expérimentation, plusieurs jeux de paramètres ont donné les meilleurs résultats (100% de précision et meilleur rappel). Pour sélectionner le jeu de paramètres final, la stratégie appliquée a été de choisir le plus grand fichier d'entraî-

TABLE 4.8 – Détail du score de détection par classes d'attaque (+ : Nombre de fichiers détectés comme normaux, - : Nombre de fichiers détectés comme anormaux)

(a) Classement des fichiers de données normaux (46 au total)

	N° 1		N° 2		N° 3		N° 4		N° 5		Taux de faux positifs
	+	-	+	-	+	-	+	-	+	-	
Normal (10s)	20	0	21	0	19	0	20	0	21	0	0%
Normal (15s)	4	0	5	0	4	0	4	0	5	0	0%
Normal (20s)	14	0	13	1	16	0	15	0	13	0	1.54%
Normal (25s)	4	0	2	0	3	0	3	0	4	0	0%
Normal (30s)	4	0	4	0	4	0	4	0	3	0	0%

(b) Classement des fichiers de données contenant une attaque (162 au total)

	N° 1		N° 2		N° 3		N° 4		N° 5		Taux de vrais positifs
	+	-	+	-	+	-	+	-	+	-	
Attaque (MIA)	3	15	2	17	2	17	2	18	2	16	88.23%
Attaque (MI)	5	13	3	17	2	15	0	19	2	15	86.74%
Attaque (MS_1)	2	16	1	16	1	17	1	16	2	17	92.21%
Attaque (MS_2)	3	16	1	16	1	15	0	17	4	14	89.97%
Attaque (MR)	5	13	5	11	6	13	5	13	5	11	70.07%
Attaque (MV)	4	13	4	15	6	12	6	13	6	13	71.79%
Attaque (AI)	3	16	4	14	4	14	4	14	4	14	79.06%
Attaque (SI)	0	18	1	17	1	18	1	17	0	20	96.73%
Attaque (SF)	0	17	0	18	0	18	2	14	0	17	97.50%

nement, puis la plus grande taille de séquences, et enfin la tolérance maximale. C'est pourquoi la taille de séquence choisie a été systématiquement 3 pour chaque expérimentation, et que la durée du fichier d'entraînement varie entre 25 et 30 secondes. Concernant les résultats de l'évaluation, la précision reste très bonne (>99%), seul un cas d'expérimentation ayant relevé un faux positif (*N°2*). Par contre, le rappel est moins bon que celui obtenu pendant la phase de test, tout en restant significativement élevé (entre 84.57% et 87.65% pour la phase d'évaluation, contre un score de 94.44% à 100% sur la phase de test). Ce résultat peut s'expliquer par le peu d'exemples utilisés pour la phase de test comparé au nombre d'exemples utilisés pour la phase d'évaluation.

La Table 4.8 détaille les résultats obtenus à l'évaluation, par classe de fichier, pour chacune des 5 expérimentations. Les fichiers normaux ont été détaillés en fonction de leur durée (Table 4.8a), tandis que les fichiers d'attaque ont été détaillés en fonction de l'opérateur d'attaque utilisé pour les générer (Table 4.8b).

Deux tendances ressortent des résultats moyens de détection, par type d'opérateur d'attaque. D'abord, les attaques basées sur les opérateurs de modification de registre (*MR*) et de modification de valeur en mémoire (*MV*) affichent le taux de détection le plus bas, suivi par l'opérateur de duplication d'instruction (*AI*). Ces

performances peuvent être expliquées par un manque d'impact de l'attaque, couplé avec un très faible nombre d'instructions supplémentaires injectées pour réaliser l'attaque (moins de 30 instructions assembleur). Également, le niveau d'observation utilisé ici (les appels API avec leur type et leur horodatage) n'est peut-être pas le plus adapté pour repérer des modifications dans la zone de données de l'application ou dans ses registres. L'observation de l'utilisation de la mémoire pourrait être un bon complément pour améliorer ces résultats.

Deuxièmement, les attaques basées sur les opérateurs de suppression d'instruction et de suppression de fonction sont très bien détectées (respectivement 96.73% et 97.50% des attaques sont repérées). En effet, ces opérateurs impactent fortement le fonctionnement de l'application, d'un point de vue temporel (pour l'opérateur *SF*) et d'un point de vue fonctionnel (pour les deux opérateurs *SI* et *SF*).

4.4 Conclusion

Ce chapitre a présenté une implémentation complète de l'approche et de l'outil d'injection d'attaque décrits dans le Chapitre 3. L'implémentation de l'approche se base sur la collecte de données réelles observées au travers de l'instrumentation d'une application avionique exécutée sur un calculateur avionique réel (*moniteur de SDA*). Ces données sont ensuite traitées hors du calculateur, sur une station de prototypage, afin de définir au mieux le SDA au travers des étapes de *prétraitement des données*, *modélisation du SDA*, et *vérification du SDA*. Un *outil d'analyse de l'utilisation des ressources* a également été développé pour proposer des pistes de réflexion quant à la réutilisation des informations disponibles dans le processus de développement IMA actuel, pour réaliser l'étape d'**Analyse statique de sécurité**.

Concernant l'implémentation de l'*outil d'injection d'attaque*, 9 opérateurs d'attaque ont été implémentés afin de représenter les trois stratégies de mutation de code définies au Chapitre 3. Ces 9 opérateurs cherchent également à représenter différentes possibilités en terme d'injection d'attaque, en intégrant pour certains les moyens d'ajouter du code et/ou de contrôler le moment d'activation de la charge malveillante (en terme temporel et/ou fréquentiel). L'outil proposé est capable de conserver l'exécution temps-réel à l'intérieur du calculateur cible, s'il n'interagit pas avec l'extérieur (pilote ou I/O). Sa configuration lui permet d'être facilement adaptable à d'autres applications. L'implémentation des opérateurs a été réalisée pour une architecture cible PowerPC. Cependant, le concept des opérateurs reste le même d'une architecture à une autre. Le portage sur une autre architecture est également facilité par l'utilisation de la liaison client-serveur GDB, qui est standardisée.

Deux types d'expérimentations viennent compléter la description de l'implémentation de l'approche. La première a démontré la pertinence de l'outil d'injection d'attaque pour le calibrage d'un HIDS efficace. En effet, cette expérimentation a montré que l'on est capable de calibrer un HIDS sans exemple d'attaque réel, qui soit efficace face à des attaques ayant un réel impact sur l'application cible. La deuxième expérimentation va plus loin en recherchant des tendances relatives à l'efficacité de

détection de l'HIDS en fonction des opérateurs d'attaque implémentés. En particulier, ces tendances pourraient permettre de réorienter les choix des caractéristiques à observer dans le SDA. Par exemple, sur l'application étudiée pendant cette seconde expérimentation, les attaques relatives aux modifications dans la mémoire ou les registres sont les plus difficiles à repérer. Il pourrait donc être intéressant d'ajouter un capteur pour observer l'utilisation de la mémoire par l'application.

Pour conclure, l'implémentation de l'approche proposée dans ce chapitre s'est concentrée sur les activités liées à la phase d'intégration. Dans ce cadre, cette implémentation n'a pas été réalisée entièrement sur un calculateur avionique réel, mais elle permet néanmoins de valider l'efficacité de détection de l'HIDS à partir de données réelles. Le prototype utilisé pour la phase d'intégration étant maintenant complètement défini, le chapitre suivant propose d'évaluer différentes solutions d'HIDS et d'implémenter sur calculateur les solutions les plus prometteuses.

Chapitre 5

Implémentation de l'HIDS embarqué pour l'évaluation des performances

Sommaire

Ce chapitre aborde la partie opérationnelle de l'HIDS, et notamment la partie qui sera embarquée à bord d'un calculateur avionique. Dans un premier temps, il est important de bien choisir les différents éléments de l'HIDS, à savoir les données observées et l'algorithme utilisé pour les traiter. C'est l'objectif des travaux présentés dans la première partie de ce chapitre. Plus précisément, plusieurs alternatives sont proposées, chacune présentant ses avantages et inconvénients, puis elles sont évaluées en terme d'efficacité grâce à l'utilisation du prototype lié aux activités d'intégration. La deuxième partie de ce chapitre présente l'implémentation de deux solutions HIDS embarquées, qui ont permis d'évaluer précisément les ressources nécessaires pour la partie embarquée de l'HIDS.

5.1 Alternatives étudiées

Dans le Chapitre 4, une première solution basée sur l'observation de séquences d'appels API ARINC 653 avec leur durée, modélisées sous la forme d'un *One-Class*

Support Vector Machine (OCSVM), a été proposée. Cette solution a permis de développer un premier prototype pour les activités de l'approche liées à l'intégration, et de valider l'intérêt d'une telle approche d'HIDS avionique. Également, cette première solution a permis de valider l'intérêt de l'outil d'injection d'attaque.

Cependant, cette solution présente une possible limitation pour être embarquée dans un calculateur en vol. Par exemple, si l'application observée effectue des appels API en boucle, la fenêtre de logs nécessaire pour conserver l'ensemble des traces émises par l'application durant un cycle d'exécution peut être très large. Dans cette section, plusieurs alternatives sont donc proposées dans le but d'étudier des variantes plus adaptées à ce type de contraintes. Les deux axes étudiés sont notamment la réduction du nombre de logs, et une mise en forme permettant la génération d'un log de taille fixe à chaque cycle d'exécution.

Le choix de l'algorithme utilisé est également remis en cause. En effet, le modèle OCSVM est plus adapté pour généraliser et représenter des données avec des valeurs continues, plutôt que la présence ou l'absence d'une caractéristique. En ce sens, il est très adapté pour représenter la durée des séquences, mais il l'est moins pour représenter l'identifiant de la séquence considérée. Pour cela, l'utilisation du *one-hot encoding* est nécessaire, mais entraîne une augmentation de la taille des données pré-traitées. D'après les conclusions de l'état de l'art présenté dans le Chapitre 2, il parait important de tester également les automates temporisés, qui semblent parfaitement adaptés au problème considéré.

5.1.1 Description des alternatives

La Table 5.1 présente les différentes alternatives étudiées dans ce chapitre. Concernant les données observées, l'alternative **Comms** (observation des services de communication uniquement) permet de réduire le nombre de logs générés, et offre une plus grande flexibilité pour implémenter le moniteur de SDA (par exemple, au niveau des drivers). Les alternatives **API_freq** et **Seq_freq** permettent d'avoir un log de taille fixe en relevant des statistiques sur un cycle d'exécution, prenant en compte ou non la notion de séquence et de durée. Pour les modèles utilisés, outre les modèles **OCSVM** et **Automate temporisé** déjà évoqués, un modèle d'**Automate simple** a également été envisagé. Celui-ci ne prend pas en compte la notion de durée, mais propose une alternative très simple aux deux autres modèles.

Finalement, sept solutions combinant ces données et ces modèles ont été sélectionnées pour être évaluées. Ces sept solutions sont décrites dans la Table 5.2, et leur implémentation est précisée dans la suite de cette section. Toutes les combinaisons n'ont pas été effectuées. En particulier, les modèles d'automates ne sont pas très adaptés pour traiter des données statistiques (API_freq et Seq_freq), et le modèle d'automate simple n'a pas été testé sur les services de communications uniquement vu la perte d'information significative que cela engendre.

TABLE 5.1 – Description des alternatives de données observées et de modèles utilisés

	Nom	Caractéristiques	Description
Données	Tous (**All**)	Information la plus complète	Observation de l'ensemble des appels API ARINC 653 effectués par l'application, avec leur horodatage
	Communications (**Comms**)	Moins de logs, implémentation plus flexible	Observation des appels API liés aux services de communication uniquement, avec leur horodatage
	Fréquences des appels API (**API_freq**)	Taille de log fixe, très simple	Observation du nombre d'appels API effectués, par type d'appel, sur un cycle d'exécution
	Fréquences des séquences (**Seq_freq**)	Taille de log fixe, prise en compte de la durée	Observation du nombre de chaque séquence d'appels API effectuée, avec sa durée moyenne, minimale, et maximale
Modèle	**OCSVM**	Généralisation des données	*One Class Support Vector Machine* : sélection des vecteurs support pour l'apprentissage et calcul de distance pour la vérification
	Automate Simple (**AS**)	Adapté aux données séquentielles	Définition des transitions autorisées pour l'apprentissage, et détection des transitions non autorisées pour la vérification
	Automate Temporisé (**AT**)	Adapté aux données séquentielles, prise en compte de la durée	Définition des transitions autorisées avec leur durée pour l'apprentissage, et détection des transitions non autorisées ou avec une durée non autorisée pour la vérification

TABLE 5.2 – Description des solutions alternatives étudiées

Nom	Données	Modèle
OCSVM_all	Tous les appels API	OCSVM
AS_all	Tous les appels API	Automate
AT_all	Tous les appels API	Automate Temporisé
OCSVM_comms	Appels API de communication	OCSVM
AT_comms	Appels API de communication	Automate Temporisé
OCSVM_API_freq	Fréquence des appels API	OCSVM
OCSVM_seq_freq	Fréquences des séquences	OCSVM

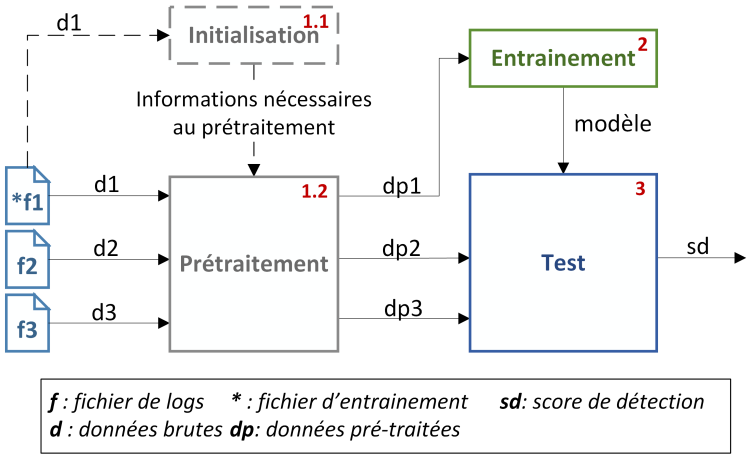


FIGURE 5.1 – Processus d'entraînement et de test

5.1.2 Détail de l'implémentation de chaque solution

Chaque solution est implémentée selon un processus similaire à celui utilisé pour les expérimentations du chapitre précédent. Ce processus est rappelé par la Figure 5.1. Dans cette section, l'implémentation des phases d'initialisation, de pré-traitement, d'entraînement et de test est détaillée pour chaque solution envisagée. Ces solutions ont été implémentées sur la station de prototypage, elles correspondent donc au prototype développé pour la phase d'intégration.

Dans tous les cas, les fichiers de logs utilisés sont les mêmes, ils ont donc le format suivant, la première colonne représentant un ID d'appel API, et la deuxième colonne son horodatage :

```
12, 4003
14, 4451
1,  4808
7,  5133
8,  6107
...
```

5.1.2.1 Phase d'initialisation et de pré-traitement

Cette partie détaille les tâches d'initialisation et de pré-traitement réalisées sur les fichiers de données brutes pour chacune des alternatives.

OCSVM_all, OCSVM_comms : Pour ces deux solutions, la phase d'initialisation consiste à retrouver l'ensemble des séquences possibles de taille n à partir du fichier qui sera utilisé pour l'entraînement, et de leur assigner un identifiant unique. Elle permet également de connaître les valeurs de durée minimale et maximale possible, utilisées pour le *scaling*. Ensuite, la phase de pré-traitement s'applique à chaque fichier indépendamment, selon une fenêtre glissante de taille l, et génère des

données pré-traitées composées de 1) l'identifiant de la séquence sous une forme *one-hot encoding* et 2) la durée de la séquence ramenée entre 0 et 1 selon les durées minimale et maximale observées à l'initialisation. Chaque donnée pré-traitée correspond donc à une séquence et contient $n + 1$ valeurs : n valeurs pour représenter l'identifiant de la séquence en format *one-hot encoding*, et 1 valeur pour représenter la durée de la séquence. Dans le cas de la solution **OCSVM_comms**, un filtrage est effectué au préalable sur chaque fichier pour ne conserver que les logs relatifs aux appels API de communication. Cette phase de pré-traitement des données a été présentée en détails dans la Section 4.1.4, qui décrit la première implémentation de l'HIDS. Si l'on reprend les données de log présentées en exemple au début de cette section, le format des données après la phase de pré-traitement (donc, après application de la mise sous forme de séquence, du *one-hot encoding*, et du *scaling*) est le suivant :

```
[0, 1, 0, 0.42]
[0, 0, 1, 0.77]
```

Sur cet exemple, il existe donc 3 séquences possibles (les trois premières valeurs de chaque vecteur), et la dernière donnée représente la durée de la séquence rapportée entre 0 et 1.

AS_all : Seule la phase de pré-traitement existe pour cette solution. Chaque fichier de données brutes est parcouru indépendamment avec une fenêtre glissante de taille n, n étant la taille de séquence considérée. Chaque donnée pré-traitée est composée des n identifiants d'appels API de la séquence considérée. Si l'on considère des séquences de taille 4, les données de log présentées en début de section seront donc pré-traitées sous la forme suivante :

```
[12, 14, 1, 7]
[14, 1, 7, 8]
```

AT_all, AT_comms : Pour ces deux solutions, le même pré-traitement que pour la solution **AS_all** est effectué, mais la durée de chaque séquence est également conservée. Chaque donnée pré-traitée est donc composée de $n + 1$ valeurs : les n identifiants des appels API de la séquence, et 1 valeur pour la durée de la séquence. Le format des données pré-traitées pour cette solution est le suivant :

```
[12, 14, 1, 7, 1130]
[14, 1, 7, 8, 1656]
```

OCSVM_API_freq : Cette solution utilise une phase d'initialisation au préalable. Cette phase d'initialisation permet de retrouver l'ensemble des m appels API différents qui sont exécutés légitimement par l'application, et de leur affecter un index entre 0 et $m-1$. La phase de pré-traitement s'effectue en deux étapes. D'abord,

le fichier de données brutes est découpé en cycles d'exécution, c'est-à-dire que les logs qui ont été générés pendant un même cycle d'exécution sont regroupés ensemble. Ensuite, une seule donnée pré-traitée est générée pour chaque groupe de logs. Cette donnée pré-traitée est composée de m valeurs, correspondant chacune au compteur d'occurrences d'un type d'appel API légitime. Pour l'application IHM-DA utilisée précédemment, si l'on considère les 18 types d'appels API différents réalisés après la phase d'initialisation de l'application, on obtient par exemple des données sous cette forme :

```
[1, 0, 9, 0, 0,  0, 0, 1,  0, 0,  8,  0,  4, 0, 1, 1, 0, 9]
[2, 1, 0, 0, 1,  8, 2, 2,  2, 0,  0,  0,  0, 0, 0, 3, 0, 0]
[0, 0, 0, 3, 2, 14, 2, 1,  6, 7,  1, 11, 15, 0, 0, 0, 0, 0]
[0, 0, 0, 3, 2, 14, 2, 1, 23, 7, 19,  0,  4, 3, 0, 0, 0, 0]
```

OCSVM_seq_freq : Pour cette solution, la phase d'initialisation permet de définir l'ensemble des p séquences légitimes de taille n à partir du fichier d'entraînement, de leur affecter un index entre 0 et $p-1$, et de retrouver les valeurs nécessaires à la normalisation des données (*scaling*). La phase de pré-traitement suit également deux étapes, comme pour la solution **OCSVM_API_freq**. La première étape est similaire et consiste à découper le fichier de données brutes en groupes de logs, chacun correspondant à un cycle d'exécution. De la même manière, une seule donnée pré-traitée est générée par cycle d'exécution. Pour cette solution, la donnée pré-traitée est composée de $4 * p$ valeurs. Pour chaque indice $i \in [0, p-1]$, e.g. pour chaque séquence possible, 4 valeurs sont donc générées : le nombre d'occurrences de la séquence dans le cycle d'exécution, et les durées minimale, maximale et moyenne observées dans le cycle d'exécution, pour cette séquence. L'exemple suivant est un extrait des données pré-traitées de l'application IHM-DA, qui présente une trentaine de séquences de taille 2 possibles :

```
[18,207,27239,3544,  2,214,231,222,  19,675,6200,1744,  0,0,0,0, ...]
[18,215,27211,3544,  2,212,216,214,  15,678,2484,1312,  0,0,0,0, ...]
[18,208,27207,3536,  2,200,222,211,  19,676,6184,1740,  2,281,313,297, ...]
[18,190,27141,3533,  2,215,220,217,  13,672,2489,1298,  1,299,299,299, ...]
```

Ces données sont ensuite reportées entre 0 et 1 selon un *scaling* min/max. Pour l'exemple précédent, cela se traduit par le format de données suivant :

```
[0,.74,.65,.85,  0,.61,.62,.60,  1,.61,.99,.92,    0,0,0,0, ...]
[0,.89,.54,.85,  0,.58,.38,.45,  .33,.67,.01,.05,  0,0,0,0, ...]
[0,.76,.53,.61,  0,.39,.48,.39,  1,.63,.99,.91,    1,.83,.93,.88, ...]
[0,.43,.26,.52,  0,.63,.44,.51,  0,.56,.01,.02,    .5,.89,.89,.89, ...]
```

Dans cet exemple, la première valeur du vecteur avant *scaling* vaut systématiquement 18. Dans ce cas particulier, la phase de *scaling* effectue donc uniquement une translation pour cette composante du vecteur de données ($valeur_{finale} = valeur_{initiale} - valeur_{normale}$, avec $valeur_{normale} = 18$ dans cet exemple).

5.1.2.2 Phase d'entraînement

Dans cette partie, les données pré-traitées présentées précédemment sont utilisées comme données d'entrée. Plus précisément, ces données ont été stockées sous forme de fichiers appelés ici *fichier de données pré-traitées*. Chaque fichier contient une liste de données pré-traitées. Chaque donnée pré-traitée (e.g. une ligne d'un fichier de données pré-traitées) sera appelé par la suite *vecteur* ou *vecteur de données*.

L'implémentation de la phase d'entraînement dépend uniquement du modèle choisi. Cette partie détaille cette phase, qui est réalisée sur un seul fichier de données pré-traitées correspondant à une exécution normale. En effet, un fichier de données contient de nombreux exemples du comportement périodique de l'application, et est donc suffisant pour modéliser son comportement normal. Cela permet également de limiter la surcharge de travail engendrée pour l'intégrateur de calculateur.

OCSVM : Le principe d'apprentissage pour obtenir un modèle OCSVM, décrit dans [Scholkopf 2001], a déjà été présenté dans la Section 4.1.5. De façon générale, le but de l'algorithme d'apprentissage est de construire un hyperplan séparant les données d'entraînement de l'origine du système considéré. Cet hyperplan est caractérisé par des vecteurs supports, sélectionnés à partir des données d'entraînement. En particulier, quatre noyaux sont couramment utilisés pour définir l'aspect général du modèle : linéaire, polynomial, fonction de base radiale (RBF), et sigmoïde. Ici, le noyau RBF est utilisé systématiquement, du fait de sa capacité à représenter des données non linéaires.

Automate simple : Ce modèle est construit comme la liste exhaustive des séquences différentes observées dans les données d'apprentissage. Cette liste correspond à l'ensemble des transitions de l'automate généré, qui décrit le comportement normal de l'application.

Automate temporisé : Ce modèle est une extension au modèle d'**Automate simple** décrit précédemment. Dans ce modèle, une liste d'intervalles acceptables de durée [min,max] est associée à chaque transition. La construction de ces intervalles est réalisée en trois étapes :

1. Les durées observées dans les données d'apprentissage sont séparées en p groupes, chaque groupe correspondant à une séquence précise.
2. Sur chaque groupe de durées, l'algorithme *k-means* est appliqué à l'aide de la librairie *scikit-learn* pour définir s sous-groupes. Les durées ayant une seule dimension, si $duree_a$ et $duree_b$ sont dans le même sous-groupe, alors toute durée qui a une valeur entre $duree_a$ et $duree_b$ sera également dans ce sous-groupe. En particulier, les sous-groupes sont donc ordonnés de façon à ce que $max(SousGroupe_i) < min(SousGroupe_{i+1}), \forall i \in [1, s-1]$. A l'issue de cette étape, on obtient donc un ensemble de s intervalles pour chaque séquence. Chaque intervalle est constitué des valeurs de durée minimale et maximale d'un sous-groupe.
3. Afin de réduire les faux positifs, une marge de *tolérance* est ajoutée aux s intervalles précédents. Les intervalles se transforment donc de la forme

$[duree_{min_i}, duree_{max_i}]$ vers la forme $[duree_{min_i} - tolérance, duree_{max_i} + tolérance]$. Les intervalles sont ensuite fusionnés s'ils se recoupent (c'est-à-dire, si $max(SousGroupe_i) >= min(SousGroupe_{i+1})$.

Pour les applications IHM-DA et IHM-DV, nous avons fixé $s = 10$ après analyse des distributions de durées pour chaque séquence produite par ces applications. Ce nombre a été volontairement choisi pour être largement suffisant, l'analyse des distributions de durées ayant montré un maximum de 4 groupes de durées pour certaines séquences. Après application de la marge de *tolérance* et fusion des intervalles, on obtient finalement entre 1 et 4 intervalles de durée pour chaque séquence du modèle.

5.1.2.3 Phase de test

Lors de la phase de test, chaque fichier de données pré-traitées (excepté celui utilisé pour l'apprentissage) est considéré individuellement, pour déterminer s'il contient ou non une attaque. Pour ce faire, chaque vecteur du fichier est évalué selon le modèle utilisé comme normal ou anormal.

Toutes les occurrences d'anomalies sont conservées dans une liste, sous la forme d'un horodatage. Cet horodatage est directement récupéré depuis les données de logs, et correspond à l'horodatage du dernier log ayant constitué l'anomalie. Pour les données séquentielles, l'horodatage de l'anomalie correspond à l'horodatage du dernier log de la séquence. Pour les données fréquentielles, l'horodatage de l'anomalie correspond à l'horodatage du dernier log du cycle d'exécution détecté comme anormal.

Ensuite, cette liste d'anomalies est parcourue pour déterminer si le nombre d'anomalies autorisées pendant 10 cycles d'exécution est atteint. Ce nombre est appelé ici le *seuil*. Si ce *seuil* est atteint, le fichier est prédit comme anormal, si ce n'est pas le cas, il et prédit comme normal. Dans ce procédé, le rôle du modèle défini précédemment est donc de donner une prédiction au niveau des vecteurs de données uniquement.

OCSVM : Pour déterminer si un vecteur de données appartient au modèle OCSVM ou non (e.g. est normal ou anormal), on utilise la fonction *decision_function* de la librairie *scikit-learn* (voir Section 4.1.7). Cette fonction calcule la distance entre le vecteur à évaluer et le modèle OCSVM. Plus précisément, elle calcule une somme de distances pondérées entre le vecteur à évaluer et les vecteurs supports, auquel est ajoutée une valeur constante négative *rho* (elle représente la distance minimale à obtenir pour que le vecteur soit considéré comme normal). On ajoute également une constante positive *tolérance* au résultat de ce calcul. Si le résultat final est négatif, le vecteur est considéré comme anormal. Si le résultat est positif, le vecteur est considéré comme normal.

Automate simple : Pour déterminer si un vecteur de données pré-traitées est normal ou anormal, il suffit de regarder s'il appartient à la liste des transitions autorisées du modèle d'automate simple construit précédemment. Si c'est le cas, le

vecteur est considéré comme normal. Si ce n'est pas le cas, il est considéré comme anormal.

Automate temporisé : Pour ce modèle, deux vérifications sont effectuées sur le vecteur à évaluer. D'abord, la séquence d'appels API doit appartenir aux transitions autorisées du modèle d'automate. Ensuite, la durée associée doit être strictement incluse dans l'un des intervalles associés à la transition considérée. Si le vecteur vérifie ces deux conditions, il est considéré comme normal. Si l'une des conditions n'est pas vérifiée, alors le vecteur est considéré comme anormal.

5.1.3 Conclusion

Cette section a présenté en détail les différentes alternatives envisagées, que ce soit pour faire varier les données observées ou le type de modèle utilisé. En particulier, l'implémentation de sept solutions différentes, résumées dans la Table 5.2, a été décrite dans cette section. L'intérêt d'implémenter et tester ces solutions est multiple : d'abord, cela permet de proposer différents compromis entre la taille de modèle à utiliser, la taille des données à observer, ou le temps de traitement de ces données. Cela permet aussi d'envisager d'autres architectures pour la solution, par exemple pour proposer une implémentation du *Moniteur de SDA* au niveau des drivers, ou pour se concentrer sur un sous-ensemble des données observées (les types d'appels API uniquement et pas le temps, par exemple, ou uniquement les services API de communication). Il est très important, dans un contexte comme celui-ci, d'envisager plusieurs solutions avant de développer un prototype embarqué. Le développement étant plus long et plus coûteux sur une cible, il est nécessaire de sélectionner au préalable la ou les solutions les plus adaptées. L'objet de la section suivante est justement de proposer des critères de sélection parmi les sept solutions présentées ici.

5.2 Évaluation de l'efficacité des alternatives

Les sept alternatives proposées ont été testées sur les deux applications IHM-DA et IHM-DV décrites précédemment. Plus précisément, les jeux de données suivants ont été utilisés pour ces deux applications :

— **IHM-DA :** 39 fichiers normaux et 43 fichiers contenant une attaque, d'une durée de 5 à 200 secondes (ce qui correspond à 100 à 4000 cycles d'exécution, pour 4000 à 689000 logs),

— **IHM-DV :** 30 fichiers normaux et 34 fichiers contenant une attaque, d'une durée de 10 à 30 secondes (ce qui correspond à 200 à 600 cycles d'exécution, pour 64000 à 215000 logs).

L'efficacité de chaque solution a été évaluée en calculant la moyenne des meilleurs scores de détection obtenus, pour 5 fichiers d'entraînement différents. L'application d'une moyenne des meilleurs scores obtenus sur différents fichiers d'entraînement permet de donner un score plus proche de la réalité. De la même façon qu'aux

TABLE 5.3 – Scores de détection pour chaque alternative de solution HIDS

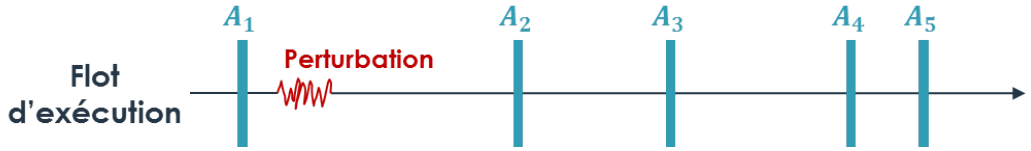
Solution	Score sur IHM-DA	Score sur IHM-DV	Moyenne
OCSVM_all	83.72%	78.18%	80.95%
AS_all	27.91%	49.09%	38.50%
AT_all	91.63%	93.94%	92.78%
OCSVM_comms	90%	77.58%	83.79%
AT_comms	93.13%	92.73%	92.93%
OCSVM_API_freq	73.95%	53.33%	63.64%
OCSVM_seq_freq	100%	90.30%	95.15%

chapitres précédents, le score de détection sélectionné ici représente la valeur de *rappel* si la *précision* est de 100%, sinon il est égal à 0. Ce score représente la capacité à détecter des attaques tout en assurant un nombre nul de fausses alertes. De la même manière que lors des expérimentations du Chapitre 4, ce meilleur score est calculé à l'aide d'une recherche exhaustive de paramètres selon un intervalle de valeurs donné pour chacun des paramètres. Selon la solution, cela représente de 4 à 36000 sets de paramètres différents testés. Pour la solution **AS_all**, le seul paramètre est la taille de séquence à considérer, c'est pourquoi seuls 4 sets de paramètres différents ont été testés pour cette solution (taille de séquence de 2, 3, 4 ou 5). Au contraire, les solutions basées sur le modèle **OCSVM** exposent de nombreux paramètres, ce qui augmente de façon exponentielle le nombre de sets de paramètres à évaluer.

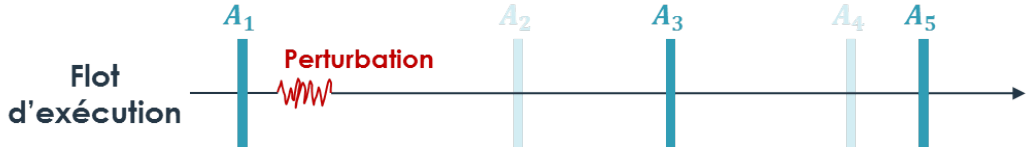
Le résultat de ces expérimentations est donné dans la Table 5.3. Pour chaque solution, le score obtenu est donné pour chaque application (IHM-DA et IHM-DV), et en moyenne sur les deux applications.

La solution **OCSVM_seq_freq** présente le meilleur résultat, en particulier sur l'application IHM-DA où pour chaque fichier d'entraînement, nous avons été capable de trouver un paramétrage permettant d'obtenir un score de 100%. Les solutions basées sur un **Automate temporisé** (**AT_all** et **AT_comms**) présentent également de très bons résultats quelle que soit l'application. Les résultats étant similaires entre les solutions **AT_all** et **AT_comms**, la solution **AT_comms** paraît plus intéressante à explorer. En effet, les données manipulées sont moins nombreuses (on observe uniquement les services ARINC 653 relatifs aux communications), ce qui devrait engendrer un plus faible impact sur l'instrumentation (*Moniteur de SDA*) et un meilleur temps de traitement pour la *partition HIDS*. La suite des expérimentations se concentre donc sur les deux solutions suivantes : **AT_comms** et **OCSVM_seq_freq**.

Les résultats similaires entre les solutions **AT_all** et **AT_comms**, et entre les solutions **OCSVM_all** et **OCSVM_comms**, montrent également qu'il n'est pas nécessaire de multiplier les points d'observation pour obtenir de bons résultats de détection. A titre d'exemple, un flot d'exécution est schématisé sur la Figure 5.2, illustrant l'exécution d'appels API $A_i, i \in [1, 5]$ et une perturbation due à une



(a) Observation de tous les appels API



(b) Observation d'un sous-ensemble d'appels API (par exemple, les services de communication)

FIGURE 5.2 – Exemple de flot d'exécution avec une perturbation due à une attaque

TABLE 5.4 – Taille de séquence utilisée pour obtenir les meilleurs résultats de détection (par fichier d'entraînement)

Application	Solution	Taille de séquence
IHM-DA	AT_comms	4,2,3,2,2
IHM-DA	OCSVM_seq_freq	2,2,3,3,4
IHM-DV	AT_comms	2,3,3,3,3
IHM-DV	OCSVM_seq_freq	5,3,2,3,2

attaque au cours de l'exécution (représentée en rouge). Dans le premier cas (Figure 5.2a, représentant une observation de tous les appels API), cette perturbation sera repérée par une anomalie sur la séquence $[A_1, A_2]$. En sélectionnant un sous-ensemble d'appels API à observer (Figure 5.2b), cette perturbation sera également repérée, mais ce sera par une anomalie sur la séquence $[A_1, A_3]$.

Enfin, une attention particulière a été portée sur la taille de séquence utilisée pour ces expérimentations. En effet, elle peut dimensionner de façon non négligeable la taille de modèle à embarquer. La Table 5.4 détaille les valeurs de ce paramètre telles qu'elles ont été obtenues dans chacun des sets de paramètres optimums, pour chaque fichier d'entraînement. Si la taille de séquence 3 est la plus représentée (9/20 cas), la taille de séquence 2 est également très représentée (8/20 cas). Pour la suite des expérimentations, le choix a été fait de travailler sur des séquences de taille 2 pour privilégier une moindre consommation de ressources.

5.3 Implémentation de la partie embarquée

Les deux solutions **AT_comms** et **OCSVM_seq_freq** ayant montré les résultats les plus intéressants en terme d'efficacité de détection, elles ont été implé-

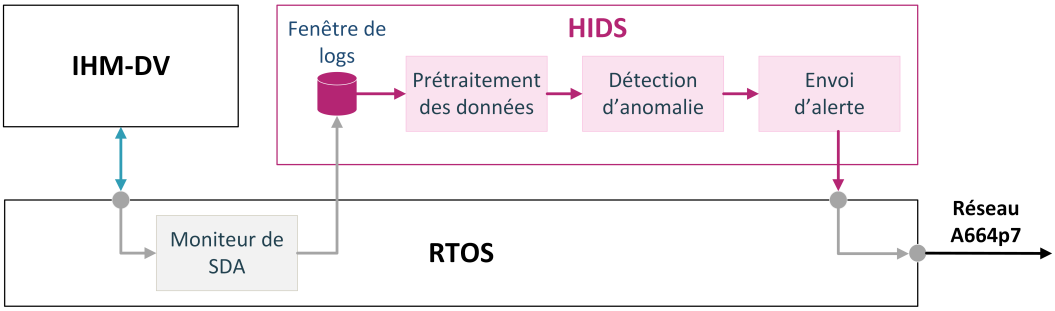


FIGURE 5.3 – Architecture du prototype d'HIDS sur cible

mentées de façon réaliste sur un calculateur avionique cible. Cette section détaille l'architecture utilisée pour implémenter l'HIDS sur une cible embarquée, et l'implémentation plus précise de chaque alternative sélectionnée.

5.3.1 Architecture implémentée sur cible

La Figure 5.3 décrit l'architecture de l'HIDS embarqué sur cible. Deux composants y sont directement intégrés : le *Moniteur de SDA* et la *Partition HIDS*.

Le *Moniteur de SDA* est similaire à la seconde version de moniteur de SDA développé pour le prototype lié à la phase d'intégration et détaillé dans le Chapitre 4. Ce moniteur est implémenté sous la forme d'un code d'instrumentation qui intercepte les appels API effectués par l'application surveillée (ici, une nouvelle version de l'application d'affichage IHM-DV), et construit les données d'observation correspondantes (qui varient en fonction de l'alternative choisie) avant de les mettre à disposition de la partition d'HIDS dans une zone mémoire spécifique, appelée ici *Fenêtre de logs*. Dans l'implémentation proposée ici, le moniteur de SDA est inséré directement dans la librairie d'appels systèmes de la partition surveillée. Cependant, ce moniteur a pour vocation d'être embarqué directement dans le RTOS, afin d'être dissocié de l'application (tel que décrit par la Figure 5.3).

Le rôle de la *Partition HIDS* est d'effectuer, à chaque cycle d'exécution, les traitement nécessaires sur les données d'observation afin de déterminer si elles sont normales ou anormales (*Pré-traitement des données* et *Détection d'anomalie*), et lever une alerte si nécessaire (*Envoi d'alerte*). Cette partition est développée et compilée sur la station qualifiée, avant d'être installée sur la cible. Le code de la partition HIDS est développé en C, et le modèle SDA est codé en dur lors du développement de la partition HIDS. D'un point de vue industriel, il sera plus intéressant d'implémenter le SDA dans une base de données dédiée. La partie de détection d'anomalie implémente la vérification du SDA telle que proposée lors de la phase d'intégration. Un outil permet de traduire automatiquement un modèle appris sur la station de prototypage, en code C correspondant à la déclaration de ce modèle. La partie d'envoi d'alerte enregistre l'ensemble des anomalies relevées sur les 10 derniers cycles d'exécution dans un buffer de taille P, P étant le nombre maximal d'anomalies autorisées pendant 10 cycles d'exécution. Si ce buffer est rem-

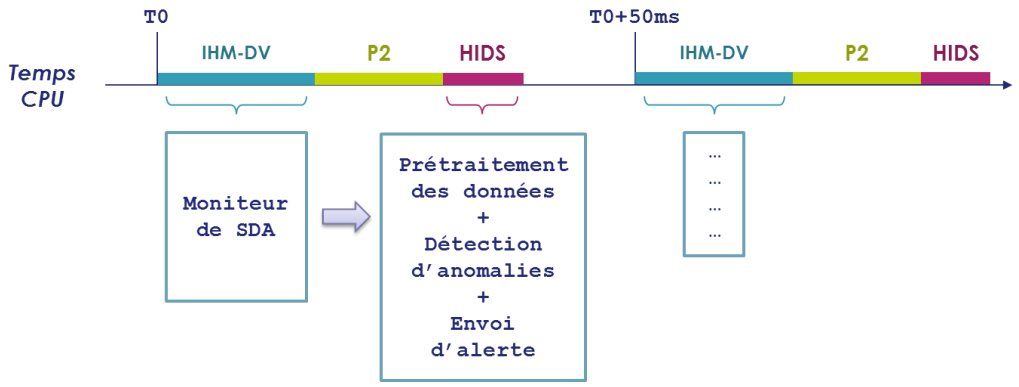


FIGURE 5.4 – Exécution temps-réel des partitions IHM-DV et HIDS

pli entièrement, cette fonction envoie un message d'alerte sur le réseau A664p7. A chaque cycle, les anomalies trop anciennes sont retirées de ce buffer.

Sur le contrôleur, on intègre un outil d'enregistrement des alertes levées par l'HIDS et envoyées via le réseau A664p7. Dans ce prototype embarqué, le rôle du contrôleur est uniquement de sauvegarder les messages A664p7 envoyés par le calculateur, pour qu'ils puissent être ensuite analysés par la station de prototypage. Initialement, les messages d'alerte ne contiennent aucune information complémentaire. Leur seul but est d'alerter sur le fait qu'un comportement anormal récurrent a été détecté par la partition HIDS. Le Chapitre 6 propose des pistes de réflexions quant aux informations supplémentaires à envoyer, afin d'aider au mieux l'analyse de l'alerte.

Sur le cas d'étude utilisé ici, les applications s'exécutent de façon temps-réel tel que décrit sur la Figure 5.4. La période des partitions IHM-DV et HIDS est de 50ms, l'application IHM-DV s'exécute pendant 6.7ms, et l'application HIDS s'exécute pendant 3.8ms. Cette durée est le temps CPU alloué à la partition, et peut ne pas être entièrement consommé à chaque cycle d'exécution. D'autres partitions s'exécutent également pendant cette période de 50ms, et sont représentées schématiquement sur la Figure 5.4 par la partition *P2*.

Si le composant d'*Envoi d'alertes* est similaire pour les deux solutions (excepté le nombre d'anomalies autorisé), les autres composants varient et sont expliqués dans les sous-parties suivantes.

5.3.2 Solution embarquée 1 : AT_comms

Pour cette solution, le *Moniteur de SDA* consiste à enregistrer un log à chaque appel API de communication effectué par l'application. Le code d'instrumentation inséré est similaire à celui présenté dans le Chapitre 4. Seule la première partie du code change : plutôt que d'exclure certains appels API selon leur identifiant, la première partie vérifie si l'appel courant est bien un service de communication. En particulier, il va vérifier si son identifiant se situe parmi les cinq valeurs suivantes :

— 42 : WRITE_SAMPLING_PORT

— 43 : READ_SAMPLING_PORT
— 47 : SEND_QUEUING_MESSAGE
— 48 : RECEIVE_QUEUING_MESSAGE
— 50 : GET_QUEUING_PORT_STATUS

Le code d'instrumentation est constitué de 35 instructions de 32-bits, il fait donc 140 octets. Pour la partie de *Pré-traitement des données* et de *Détection d'anomalie*, le modèle d'automate temporisé est constitué de 5 états identifiés par un index $i \in [0;4]$. Chaque état correspond à un identifiant d'appel API de communication $ID \in \{42, 43, 47, 48, 50\}$, qui correspond au premier appel d'une séquence. Les transitions possibles d'un état à l'autre (e.g. les séquences autorisées) sont implémentées sous la forme de listes chaînées. Pour une transition donnée, les intervalles de durée autorisés pour effectuer cette transition sont également implémentés sous la forme d'une liste chaînée. Ce modèle d'automate temporisé est implémenté en C à l'aide de trois structures (Automate temporisé, Transition, et Intervalle), tel que présenté par le Listing 5.1.

La structure *AUTOMATE_TEMPORISE* représente un automate constitué de *nb_etats* états (ici, 5), d'une table de correspondance entre un identifiant d'appel API ID et son index i (*ID_table[5]*), et d'un tableau de pointeurs représentant les transitions autorisées à partir de l'état n°i (*liste_transitions[5]*). Les transitions sont représentées par la structure *TRANSITION*, contenant l'identifiant du deuxième appel API de la séquence (*ID_out*), la liste des intervalles de temps autorisés pour cette séquence (*intervalles*), et la prochaine transition autorisée à partir de l'état n°i. Enfin, la structure *INTERVALLE* contient deux entiers représentant les bornes d'un intervalle de durée autorisé ([*debut* ;*fin*]), et un pointeur vers le prochain intervalle de temps autorisé pour la séquence courante.

La partie de *Pré-traitement des données* consiste à transformer les données d'observation (enregistrées par le *Moniteur de SDA* dans la *Fenêtre de logs*) sous la forme d'un triplet $[input\_index, output\_ID, duration]$ (représentant une séquence de deux appels API avec sa durée, tel que décrit dans la section précédente). Dans cette même boucle, la *Détection d'anomalie* vérifie ensuite si ce triplet existe bien dans le modèle d'automate temporisé décrit précédemment. D'abord, l'algorithme vérifie l'existence d'une transition $[input\_index, output\_ID]$, puis vérifie que la durée (*duration*) est bien comprise entre les bornes de l'un des intervalles correspondant à cette transition. Ce procédé est décrit par le pseudo-code présenté par le Listing 5.2.

La fonction *pretraitement* correspond au *Pré-traitement des données*, la fonction de *recherche_transition* consiste à rechercher la transition $[input\_index, output\_ID]$ dans l'automate, la fonction de *recherche_intervalle* consiste à chercher si la valeur *duration* est comprise dans l'un des intervalles de durées, et la fonction *lever_anomalie* consiste à enregistrer l'horodatage du dernier log de la séquence dans un buffer d'anomalies.

La taille finale du FLS correspondant à la partition HIDS pour cette solution est de 28.4ko. A titre de comparaison, l'application surveillée a une taille de 27.2Mo.

```c
// Liste chaînée d'intervalles de temps
struct INTERVALLE{
        unsigned int debut;
        unsigned int fin;
        struct INTERVALLE * prochain;
};
// Liste chaînée de transitions de l'automate
struct TRANSITION{
        unsigned int ID_out;
        struct INTERVALLE * intervalles;
        struct TRANSITION * prochain;
};
// Automate temporisé
typedef struct{
        unsigned int nb_etats;
        unsigned int ID_table[5];
        struct TRANSITION* liste_transitions[5];
} AUTOMATE_TEMPORISE;
```

Listing 5.1 – Structures utilisées pour représenter un Automate Temporisé (Code C)

```python
# Exemple de deux logs
log1 = (42,100)
log2 = (43,200)
# Pré-traitement des données log1, log2
# input_index=0, output_ID=43, duration=100
(input_index, output_ID, duration) = pretraitement(log1,log2,ID_table)
# Détection d'anomalie
# 1. Est-ce que la transition (42,43) existe ?
transition_courante =
↪  recherche_transition(liste_transitions[input_index],output_ID)
if transition_courante==NULL :
  lever_anomalie(log2.timestamp)
else :
  # 2. Est-ce que la valeur 100 est dans un intervalle de la transition
  ↪  (42,43)?
  intervalle_courant =
  ↪  recherche_intervalle(transition_courante.intervalles,duration)
  if intervalle_courant==NULL :
    lever_anomalie(log2.timestamp)
```

Listing 5.2 – Prétraitement des données et détection d'anomalie pour la solution **AT_comms** (Pseudo-code)

La taille de l'application HIDS, qui contient notamment le modèle spécifique à l'application surveillée, est donc de 0.11% de la taille de l'application surveillée. Cette taille est suffisamment faible pour que le code de l'HIDS soit embarquable.

5.3.3 Solution embarquée 2 : OCSVM_seq_freq

Pour cette solution, le *Moniteur de SDA* est un peu plus compliqué. En effet, à chaque appel API effectué par l'application IHM-DV, le moniteur doit mettre à jour la fenêtre de logs et pas seulement y ajouter une valeur. Plus précisément, le code d'instrumentation effectue les actions suivantes :

1. Vérifier si c'est le premier appel API du cycle d'exécution
2. Mettre à jour la fenêtre de logs
   (a) Calculer la séquence et la durée courantes
   (b) Retrouver l'adresse dans la fenêtre de logs correspondant à la séquence courante
   (c) Mettre à jour les durées min, max et moyenne
   (d) Mettre à jour le compteur
3. Sauvegarder les valeurs (ID appel, Horodatage) courantes
4. Continuer l'exécution normale

Pour cette solution, le code d'instrumentation comporte 64 instructions sur 32-bits, soit une taille de code de 256 octets. Si les calculs effectués par le *Moniteur de SDA* sont plus importants pour cette solution que pour la solution **AT_comms**, ils restent finalement largement raisonnables en terme de taille mémoire. Cependant, la fenêtre de logs implémentée dans la partition HIDS a une taille beaucoup plus grande, de $53 \times 53 \times 4 \times 4 = 44944$ octets. En effet, elle correspond à une fenêtre exhaustive des séquences pouvant exister, la valeur d'un identifiant variant entre 0 et 52. Ainsi, l'ensemble des séquences de deux appels nécessite de réserver 53×53 emplacements mémoire, chacun contenant 4 valeurs (durées min, max, moyenne et compteur), elles-mêmes codées sur 4 octets.

La phase de *Pré-traitement des données* va d'abord sélectionner les données d'observation enregistrées dans la fenêtre de logs pour ne conserver que les séquences légitimes observées à l'entraînement. Sur l'application IHM-DV, cela correspond à 45 séquences différentes de taille 2, soit $45 \times 4 = 180$ valeurs. Ces 180 valeurs sont reportées sur une échelle de 0 à 1 avec un *scaling* min/max.

La phase de *Détection d'anomalies* vérifie si le vecteur final de 180 valeurs entre 0 et 1 est bien à l'intérieur du modèle OCSVM enregistré en dur dans le code de la partition HIDS. Pour l'application IHM-DV, ce modèle est constitué de 21 vecteurs support.

La taille finale du code de l'application HIDS pour cette solution est de 63.9 ko. Cela représente donc 0.24% de la taille de l'application surveillée, ce qui est tout à fait acceptable.

5.4 Expérimentations

Pour chaque solution implémentée, deux types d'expérimentations ont été réalisées pour déterminer 1) le temps de calcul nécessaire pour le *Moniteur de SDA* et 2) le temps d'exécution réel de la *Partition HIDS*. Le déroulement et les résultats de chacune de ces expérimentations sont donnés dans cette section.

5.4.1 Temps de calcul pour le *Moniteur de SDA*

Pour évaluer le temps nécessaire au *Moniteur de SDA*, deux types de captures sont réalisées. Pour chacune d'elle, deux points d'arrêt sont définis autour du code commun à tous les appels système (première et dernière instruction), et enregistrent la valeur du registre *TBL*. Deux types de captures sont effectuées pour chaque solution : avec l'insertion du code d'instrumentation, ou sans ce code d'instrumentation. Plus de 10000 captures de durée ont été effectuées pour chaque expérimentation.

Les Figures 5.5a et 5.5b donnent la distribution de temps observée, par type d'appel API, avec ou sans instrumentation, pour chacune des solutions. Dans les deux cas, on peut observer une légère hausse de la durée.

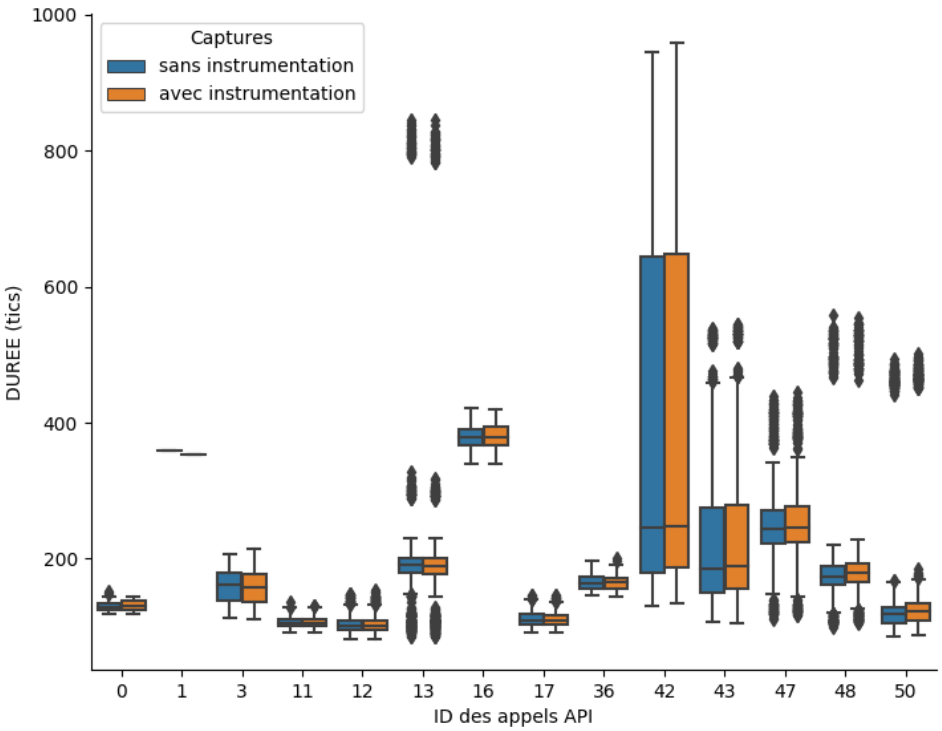
Afin de caractériser plus précisément cette hausse, les Figures 5.6a et 5.6b proposent une concaténation de ces résultats, soit en séparant les appels API de communication des autres (pour la solution **AT_comms**), soit en donnant une tendance globale sur l'ensemble des appels API (pour la solution **OCSVM_seq_freq**).

Pour la solution **AT_comms**, on observe une hausse de 5 tics d'horloge sur la valeur médiane pour les services de communication, contre une baisse de 1 tic pour les autres services (l'ajout du code d'instrumentation est donc indissociable du bruit pour ces autres appels). Pour la solution **OCSVM_seq_freq**, on observe également une légère augmentation de 5 tics d'horloge pour la valeur médiane. Cette valeur de 5 tics d'horloge correspond à 135ns, ce qui est infime. Si l'on considère la valeur médiane moyenne, cette augmentation représente environ 3% de hausse. Cette augmentation est dans les objectifs de consommation de ressources fixés au Chapitre 1 (moins de 5%).

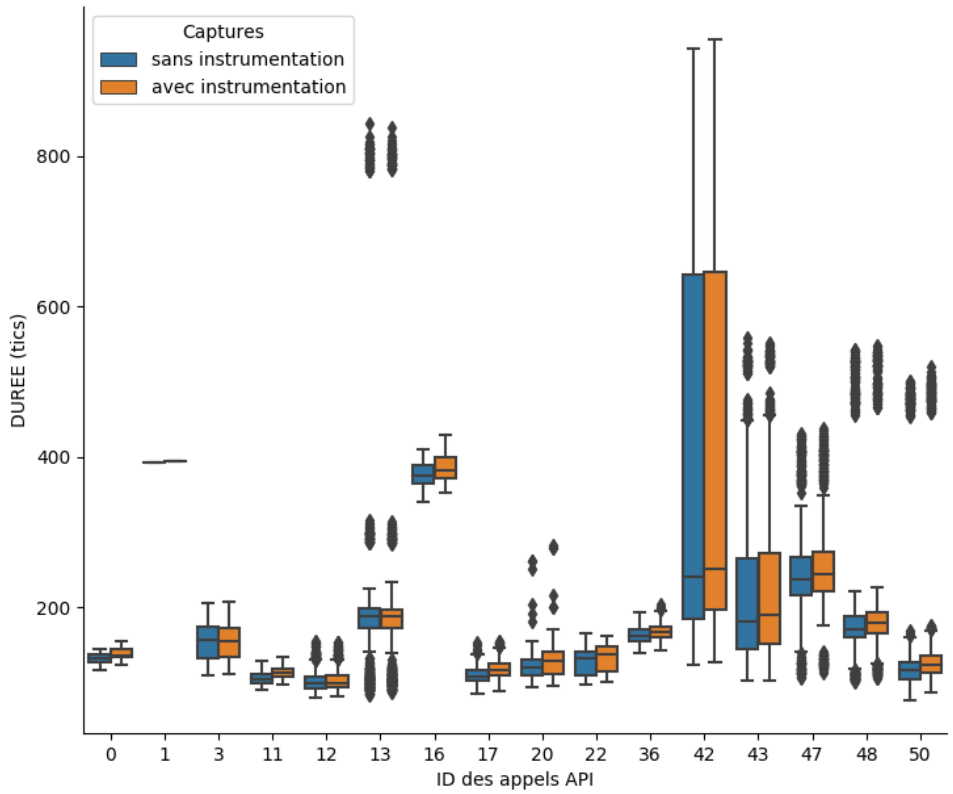
De plus, ces expérimentations ont permis de déterminer un temps minimum pour effectuer un appel API, qui est ici de 50 tics d'horloge. Cette valeur sera utilisée pour estimer le pire cas en terme de temps de calcul pour la partition HIDS dans l'expérimentation suivante.

5.4.2 Temps de calcul pour la *Partition HIDS*

Pour capturer le temps de calcul nécessaire à la partition HIDS, deux points d'arrêt ont également été utilisés autour du code de la partition. Si l'on schématise la structure de cette partition, elle commence par une phase d'initialisation puis une boucle infinie qui va s'occuper du traitement à chaque cycle d'exécution. C'est la durée de ce traitement qui est capturée ici. Cette structure et les deux points d'arrêt utilisés pour effectuer les captures de durée de la partition HIDS sont décrits par le Listing 5.3.

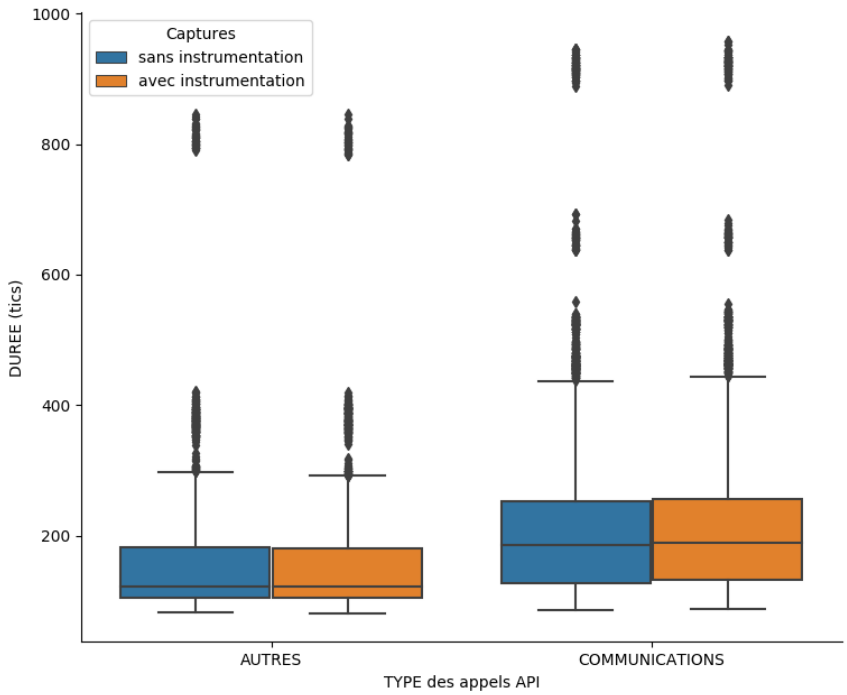


(a) Solution AT_comms

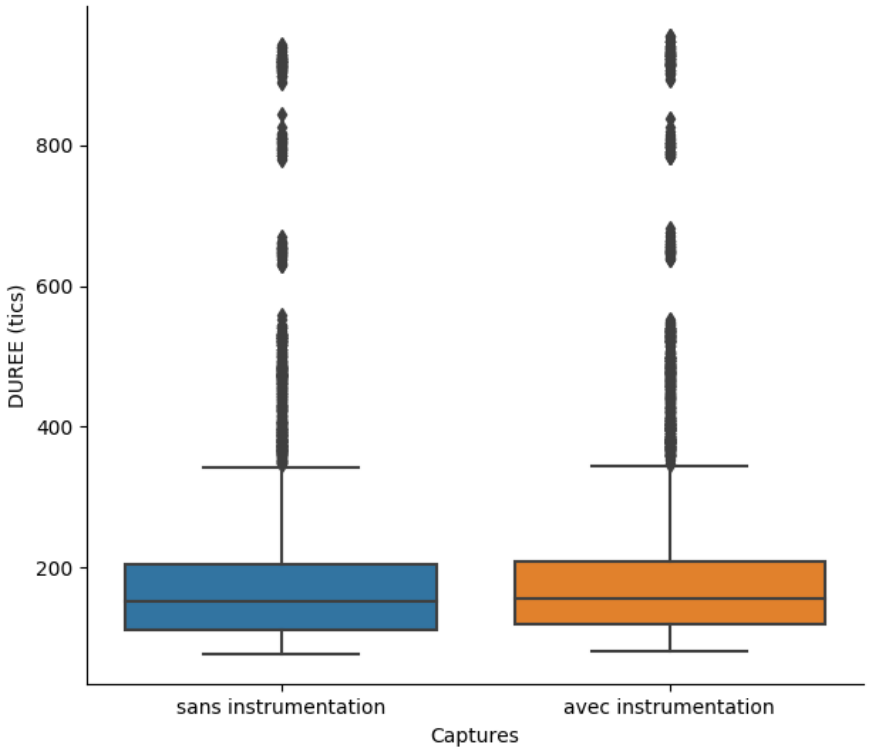


(b) Solution OCSVM_seq_freq

FIGURE 5.5 – Distribution de temps d'exécution, avec ou sans instrumentation



(a) Solution AT_comms (services de communications et autres services)

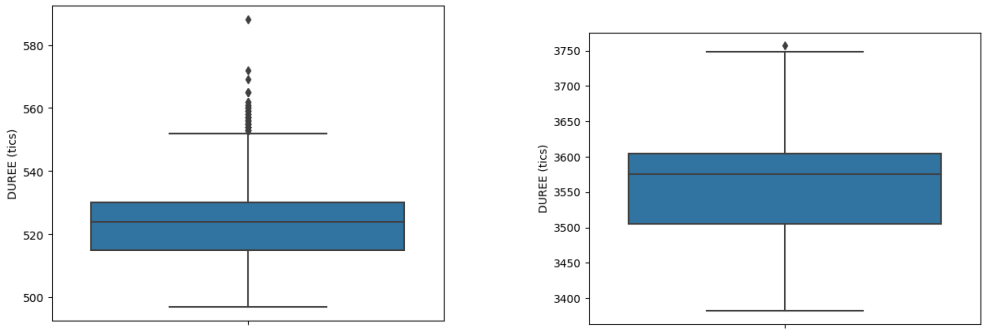


(b) Solution OCSVM_seq_freq (tous les services)

FIGURE 5.6 – Distribution globale du temps d'exécution, avec ou sans instrumentation

```c
# Phase d'initialisation de la partition
init();
# Boucle infinie
while(1) {
    # Premier point d'arrêt
    # Traitement périodique
    do_stuff();
    # Deuxième point d'arrêt
    # Attendre le prochain cycle d'exécution
    PERIODIC_WAIT();
}
```

Listing 5.3 – Structure d'une partition avionique et emplacement des points d'arrêts permettant de capturer la durée d'exécution de cette partition (Code C)



(a) Solution AT_comms (b) Solution OCSVM_seq_freq

FIGURE 5.7 – Distribution du temps d'exécution de la partition HIDS sur plus de 7000 captures

Une capture de plus de 7000 cycles d'exécution a été réalisée pour chacune des solutions (**AT_comms** et **OCSVM_seq_freq**). La distribution de durée pour chaque solution est donnée sur les Figures 5.7a et 5.7b.

Pour la solution **AT_comms**, on observe une durée d'exécution médiane de 524 tics d'horloge, ce qui correspond à 0.014ms, soit 0.21% du temps d'exécution alloué à la partition surveillée IHM-DV, ce qui est un excellent résultat. Cependant, ce résultat concerne le cas d'utilisation moyen de l'application. Pour ces captures, entre 136 et 158 logs étaient traités, pour une médiane de 140 logs traités par cycle. Si l'on considère le pire cas, l'application pourrait faire des appels API en boucle pendant toute sa durée d'exécution. Ce comportement peut être légitime pour certaines applications, par exemple, elles peuvent exécuter un service de lecture de message jusqu'à obtenir un message (la non disponibilité du message est un retour possible du service). Dans ce cas, on peut estimer le nombre maximal de logs gé-

nérés pendant 6.7ms (temps d'exécution de la partition surveillée), en considérant qu'elle peut faire au maximum un appel tous les 50 tics d'horloge. Dans ce cas, le nombre maximal de log généré est de 5025. On peut raisonnablement considérer que le temps d'exécution de la partition HIDS, pour la solution **AT_comms**, est linéairement proportionnel au nombre de logs à traiter. Dans ce cadre, on peut estimer le pire temps d'exécution à 18.864 tics d'horloge, soit 0.50ms, soit 7.5% du temps alloué à la partition surveillée, ce qui dépasse le budget alloué à la partition HIDS. Néanmoins, cette solution peut être très efficace pour des applications qui n'ont pas ce type de comportement, à condition de borner le nombre de logs possible par cycle d'exécution. Dans ce cas, un débordement du nombre de logs peut représenter en lui-même une anomalie de comportement.

Pour la solution **OCSVM_seq_freq**, la durée d'exécution médiane observée est de 3557 tics, soit 0.095ms, soit 1.42% du temps alloué à la partition IHM-DV. Cette solution respecte largement le budget alloué à la partition HIDS. Elle est moins efficace sur le cas observé que la solution **AT_comms**, cependant ce temps est garanti quel que soit le nombre d'appels API effectués par l'application surveillée. En effet, dans tous les cas, un seul log sera à traiter. Si l'on reprend le pire cas évoqué précédemment, le temps de traitement de l'HIDS, pour la solution **OCSVM_seq_freq**, restera autour de 0.095ms.

5.5 Conclusion

Dans ce chapitre, nous avons proposé différentes alternatives de solution afin de concentrer les efforts liés au portage des développements sur la cible, aux solutions les plus prometteuses. Dans ce cadre, sept alternatives ont été proposées, faisant varier 1) les données utilisées et 2) le type de modèle utilisé pour la détection d'anomalie. Concernant les données, nous avons proposé d'observer de façon séquentielle tous les appels API ou seulement ceux relatifs aux communications, ou d'observer de façon fréquentielle (sur un cycle d'exécution) les appels API effectués ou les séquences d'appels API effectuées avec leur durée. Concernant le modèle, un OCSVM, un automate, et un automate temporisé ont été utilisés, de façon à obtenir une bonne généralisation des données ou un modèle adapté aux données séquentielles plus ou moins simples.

L'efficacité de chaque solution a ensuite été évaluée sur deux applications différentes, IHM-DA et IHM-DV. La comparaison de ces résultats a permis de mettre en évidence deux alternatives très efficaces (score de détection de plus de 90% sur chaque application test), qui ont été implémentées sur cible par la suite. L'implémentation sur cible a permis d'évaluer précisément les ressources nécessaires à la mise en place de ces solutions, qui sont résumées dans la Table 5.5.

Ces résultats ont montré que la consommation de ressources de ces solutions remplit les objectifs fixés, à savoir utiliser moins de 5% des ressources disponibles, excepté pour la solution **AT_comms** dans le pire cas estimé.

Ces travaux nous ont donc permis de valider la faisabilité de l'implémentation

TABLE 5.5 – Récapitulatif des résultats obtenus (consommation de ressources sur cible)

Solution	Taille du code (.elf)	Temps de calcul		
		Moniteur de SDA (par appel)	*Partition HIDS*	*Partition HIDS* (pire cas)
AT_comms	28.4ko (0.11%)	+3%	0.21%	7.5%
OCSVM_seq_freq	63.9ko (0.24%)	+3%	1.42%	1.42%

d'un système de *Détection d'anomalie* embarqué sur un calculateur avionique. Le chapitre suivant propose des pistes pour traiter les deux autres activités liées à la phase d'opération, à savoir la *Confirmation d'attaque* et l'*Investigation au sol*.

Chapitre 6

Aide au diagnostic

Sommaire

Ce chapitre présente quelques pistes de réflexion relatives aux parties dédiées à la *Confirmation d'attaque* et à l'*Investigation au sol* de l'approche générale d'introduction d'un HIDS avionique proposée dans cette thèse. Dans les chapitres précédents, l'approche a été évaluée de façon globale au regard des exigences du domaine avionique, et plus particulièrement concernant l'efficacité de détection de la partie de *Détection d'anomalie*, et sa compatibilité à un environnement embarqué. Ce chapitre propose d'aller un peu plus loin, en proposant d'adjoindre aux alertes envoyées par la partie de *Détection d'anomalie* des informations supplémentaires, qui permettront de caractériser l'alerte d'une part (*Confirmation d'attaque*), et d'aider le diagnostic ultérieur au sol d'autre part (*Investigation au sol*).

Ce chapitre se découpe en quatre sections. La première présente le principe d'aide au diagnostic proposé dans cette thèse, et notamment l'architecture envisagée pour la réaliser. La deuxième section détaille l'implémentation de ces travaux dans

notre environnement d'étude. La troisième section décrit plusieurs expérimentations réalisées afin d'évaluer l'intérêt de cette partie d'aide au diagnostic et sa capacité à être embarquée dans un calculateur avionique. Enfin, la dernière section termine ce chapitre en résumant les pistes explorées pour réaliser l'activité d'aide au diagnostic, les conclusions associées, et les perspectives relatives à ces travaux plus prospectifs que ceux présentés aux chapitres précédents.

6.1 Principe

Les chapitres précédents se sont concentrés sur la partie de *Détection d'anomalie*, dont le but est de lever une alerte lorsqu'un comportement anormal survient. Le principe proposé ici cherche à construire un message relatif à cette alerte, qui apporte suffisamment d'information pour faciliter le traitement de l'alerte à bord d'une part, et pour aider un acteur au sol à réaliser un diagnostic suite à cette alerte d'autre part. Une première partie présente une vue globale des activités liées à la construction finale d'un message d'alerte au sein de l'HIDS. Cette approche se base sur la définition d'une signature de l'alerte, qui est ensuite comparée à une base de connaissances embarquée afin de sélectionner les informations essentielles à transmettre au travers du message d'alerte. Elle se découpe en quatre composants (Extraction de la signature, Recherche dans la base de connaissances, Construction du message d'alerte, et Investigation au sol), qui sont détaillés dans les parties suivantes.

6.1.1 Vue globale

La Figure 6.1 donne un aperçu de la solution proposée pour consolider une alerte provenant de l'entité de *Détection d'anomalie*. On introduit une entité embarquée de *Confirmation d'attaque*, dont le rôle est d'analyser les anomalies relevées par l'entité de *Détection d'anomalie* afin de caractériser l'alerte et construire le message d'alerte en conséquence. Tout d'abord, cette entité calcule des caractéristiques relatives à l'alerte levée afin de construire une signature (**Extraction de la signature**). Ensuite, cette signature est évaluée par rapport à une base de connaissances pour attacher un label à la signature (**Recherche dans la base de connaissances**). La dernière partie de **Construction du message d'alerte** prend en compte à la fois les données extraites de l'alerte et le label déterminé précédemment pour construire le message d'alerte final qui sera ensuite traité à bord, avec une éventuelle réaction (**Traitement du message d'alerte à bord**). Cette réaction après alerte est un sujet complexe qui n'est pas abordé dans ces travaux. Le rôle de l'HIDS ici est uniquement d'envoyer un message d'alerte, qui sera réceptionné et traité par une autre entité à bord (par exemple, une entité de stockage et de corrélation similaire à la fonction BITE (*Built-In Test Equipment*) utilisée pour les messages de maintenance, une entité de prise de décision et réaction automatique similaire aux moniteurs de *safety*, ou encore une entité d'affichage d'alertes et procédures au pilote similaire au *Flight Warning*). Cette entité responsable de la réception

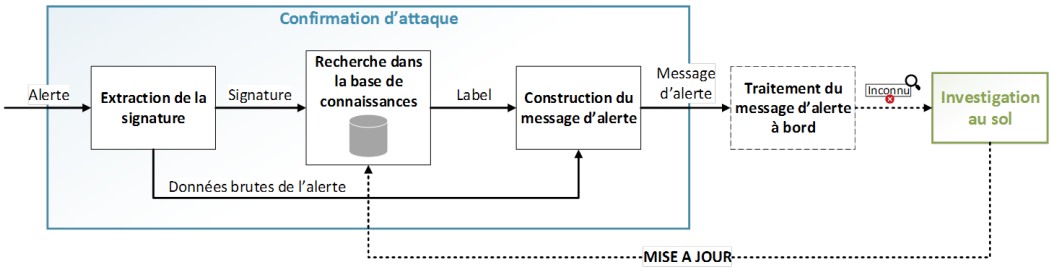


FIGURE 6.1 – Approche de consolidation du message d'alerte

et du traitement du message d'alerte à bord est représentée ici par la partie de **Traitement du message d'alerte à bord**. Finalement, pour les messages d'alerte correspondant à un label "inconnu", une *Investigation au sol* est réalisée afin de déterminer le véritable label correspondant à l'alerte levée (classe d'attaque, fausse alerte ou défaillance), et mettre à jour la base de connaissances en conséquence. Dans la mesure du possible, il reste cependant intéressant de conserver l'ensemble des messages d'alerte au sol, ceux présentant un label "inconnu", mais également ceux présentant un label reconnu dans la base de connaissances.

6.1.2 Extraction de la signature

Le but de cette fonction est d'extraire de l'information sur l'application afin de créer une signature de l'alerte. Pour ce faire, plusieurs données de nature différente peuvent être considérées. Notamment, ces données peuvent être divisées en trois catégories :

— **Données relatives à l'alerte :** Ces données représentent les informations qui découlent directement de l'alerte levée. Ici, ces données correspondent donc directement aux anomalies relevées par le détecteur d'anomalies.

— **Données d'exécution de l'application :** Ces données correspondent aux informations qui sont enregistrées pendant l'exécution de l'application sous surveillance. C'est par exemple le cas des données brutes enregistrées par le *Moniteur de SDA*. Pour le moment, ces données sont utilisées uniquement pour déterminer s'il y a des anomalies, puis elles sont supprimées. Il serait donc possible de conserver tout ou partie de ces informations, voir d'introduire de nouvelles données enregistrées par le *Moniteur de SDA*, afin de construire une signature d'alerte.

— **Données de contexte de l'application :** Cette dernière catégorie représente les données qui sont observables pendant l'exécution de la partition HIDS, c'est-à-dire entre deux cycles d'exécution de la partition surveillée.

La Table 6.1 donne plusieurs exemples d'informations pouvant entrer dans la construction d'une signature, en fonction de leur catégorie. Chaque catégorie amène un niveau différent d'information. Dans le cas de données relatives à l'alerte, les données sont directement liées aux résultats de la détection d'anomalies. Elles sont donc

TABLE 6.1 – Exemples de données pouvant entrer dans la composition d'une signature d'alerte

Nom	Catégorie	Description
Vecteur de données anormal	Alerte	La donnée brute ayant donné l'anomalie
Distance de l'anomalie au modèle	Alerte	La distance observée entre la donnée anormale et le modèle
Distribution des anomalies	Alerte	La façon dont l'ensemble des anomalies sont réparties (par exemple, les appels impactés, les séquences impactées, ou les zones de code impactées)
Registres généraux	Exécution	Valeur des registres généraux à chaque point d'observation (représentent notamment les paramètres d'appel de fonctions)
Adresse de retour	Exécution	Valeur des adresses de retour de fonction à chaque point d'observation
Contenu de la pile	Exécution	Valeurs contenues dans la pile à chaque point d'observation
Registres spécifiques	Exécution	Valeur des registres spécifiques à chaque point d'observation (par exemple, les compteurs de performance, compteur général, registre de conditions)
État des registres généraux	Contexte	Valeur des registres généraux à la fin de l'exécution de la partition (notamment, le registre pointeur d'instruction courante)
État de la configuration mémoire	Contexte	Droits d'accès enregistrés dans la MMU pour la partition surveillée
État de la pile	Contexte	Valeurs contenues dans la pile à la fin de l'exécution de la partition

très ciblées sur l'anomalie elle-même et devraient permettre de bien comprendre l'impact relatif à l'alerte. Les données d'exécution sont beaucoup plus générales et peuvent donc permettre de mieux comprendre l'origine de l'alerte, mais elles peuvent représenter une très grande quantité de données à enregistrer, ce qui n'est pas forcément adapté au contexte embarqué. Enfin, les données de contexte peuvent être observées ponctuellement par la partition d'HIDS sans impacter l'exécution temps-réel de l'ensemble. Cependant, l'information disponible est limitée puisque les partitions tâchent généralement de finir leur exécution avant la fin du temps qui leur est alloué. Dans ce cas, il peut être très difficile de déterminer précisément pourquoi une alerte a eu lieu, en étudiant uniquement l'état de la partition une fois son exécution terminée. Cependant, une déviation de cet état par rapport à un état normal peut amener des éléments clés pour un diagnostic, par exemple si l'état de la pile est très différent de l'état habituel, ou si une partie du code a été modifiée.

On a donc à notre disposition un ensemble d'informations qui pourraient aider

au diagnostic. Cependant, il est important de prendre en compte que 1) l'espace mémoire disponible à bord est limité et 2) les moments d'observation restent limités, pour réduire l'impact de l'HIDS sur le calculateur et les applications qui y sont hébergées.

Afin de sélectionner les informations parmi ces exemples, il est important de déterminer précisément l'objectif du traitement de ces données. En particulier, le but de ces travaux est de déterminer la classe d'attaque correspondant à l'alerte levée, ou de déterminer si cette alerte correspond à un cas non pris en compte lors de la modélisation du SDA (cas de défaillance ou faux positif). Il faut donc que les informations contenues dans la signature soient capables de différencier une attaque, un évènement de *safety*, et un faux positif. Ces différents cas peuvent être caractérisés comme suit :

— **Attaque :** De façon générale, une attaque devrait se caractériser par un impact spécifique (une attaque a un but précis), qui peut se traduire au travers du flot d'exécution de l'application, de ses permissions, de sa configuration, de l'utilisation de ses registres, etc.

— **Défaillance :** Dans ce cas, un impact devrait être clairement observé, mais celui-ci doit être corrélé avec des alertes des moniteurs de *safety*.

— **Faux positif :** Ce cas très rare est caractérisé par le fait qu'il n'y a aucun impact sur l'application (anomalies très proches du comportement normal, pas de liens entre elles, état général connu).

Au regard de ces trois cas, il est donc important de traduire dans la signature les 3 caractéristiques suivantes : 1) la corrélation avec une alerte relative à la *safety*, 2) l'état de l'environnement de l'application (en particulier, s'il correspond à un état connu), 3) un potentiel impact sur l'application.

Concernant la corrélation avec une alerte *safety*, plusieurs niveaux peuvent être envisagés. Tout d'abord, les informations enregistrées depuis les moniteurs *safety* peuvent être récupérés localement via le RTOS ou l'application surveillée. Les messages de maintenance peuvent également être récupérés localement au travers de la fonction *BITE*. Ensuite, une corrélation à un niveau plus global (par exemple, au niveau de la suite avionique) peut permettre de donner plus de contexte à l'alerte levée par l'HIDS (par exemple, si une suite de défaillances sur la suite avionique a mené à une modification du comportement de l'application surveillée, ayant déclenché ensuite l'alerte de l'HIDS). On prend ici l'hypothèse que cette corrélation est faite directement lors du **Traitement du message d'alerte à bord**.

L'état de l'environnement de l'application peut être représenté à l'aide des données de contexte, mais également par certaines données d'exécution comme les adresses de retour de fonction, qui peuvent par exemple démontrer une utilisation détournée du code de l'application. Dans ce cadre, nous proposons ici d'enregistrer les adresses de retour de fonction aux points d'observation (donc, à chaque appel API ARINC 653 effectué par la partition surveillée), afin de repérer une utilisation détournée du code de l'application.

Enfin, renseigner des informations relatives au potentiel impact sur l'application permet de redonner un premier sens fonctionnel à l'alerte levée. Plusieurs éléments peuvent être considérés ici. Dans un premier temps, la distance des anomalies au modèle et la variabilité entre les anomalies sont des caractéristiques clés pour déterminer un faux positif. Dans le cas où les anomalies traduisent un impact précis sur le fonctionnement de l'application (attaque ou défaillance), on peut définir plusieurs types d'impacts potentiels sur le flot d'exécution de l'application, et en extraire des caractéristiques permettant de qualifier ces impacts. Ce travail nous a permis de mettre en évidence des caractéristiques telles que la ressemblance entre anomalies, l'aspect consécutif de l'occurrence des anomalies, ou le fait d'avoir plusieurs anomalies dans un même cycle d'exécution.

Pour conclure, nous avons choisi de baser la définition d'une signature sur les éléments suivants :

— Corrélation avec une alerte *safety* (reporté au **Traitement du message d'alerte à bord**, qui ne sera donc pas traité ici)

— Enregistrement des adresses de retour de fonction lors de la collecte de données (par le moniteur de SDA, pendant l'exécution de la partition surveillée)

— Corrélation entre les anomalies (distance entre les anomalies et le modèle, ressemblance entre anomalies, occurrence d'anomalies consécutives, apparition de multiples anomalies sur un même cycle)

6.1.3 Recherche dans la base de connaissances

Une fois la signature extraite de l'alerte courante, elle est comparée à une base de connaissances, dont les entrées peuvent correspondre soit à des attaques (comme dans une base de signatures d'antivirus), soit à des alertes relatives à la *safety*, soit à des fausses alertes déjà connues.

Chaque entrée de la base de connaissance est composée d'un motif de signature avec un label associé. Pour les systèmes anti-virus, dont ces travaux se sont inspirés, les motifs correspondent généralement à une suite d'octets spécifique, tandis que le label est l'identifiant du virus associé. Dans notre contexte, nous considérons plutôt des motifs relatifs aux champs de la signature, tels que des valeurs spécifiques pour certains champs, ou des intervalles de valeur. Le label associé à un motif de signature peut correspondre à l'identifiant d'une classe d'attaque, un identifiant de défaillance, ou un identifiant de fausse alerte. Si la signature évaluée ne correspond à aucune entrée de la base de connaissance, le label attaché à cette signature sera le label "Inconnu".

Dans certains cas, et notamment en utilisant des intervalles de valeurs pour définir le motif d'une signature, il arrive qu'une même signature d'alerte puisse correspondre à plusieurs motifs (donc, plusieurs entrées) de la base de connaissances. Dans ce cas, il peut être judicieux d'enregistrer l'ensemble des labels, chacun accompagné d'une valeur de vraisemblance. Cette valeur représentera la probabilité que le label corresponde effectivement à l'alerte.

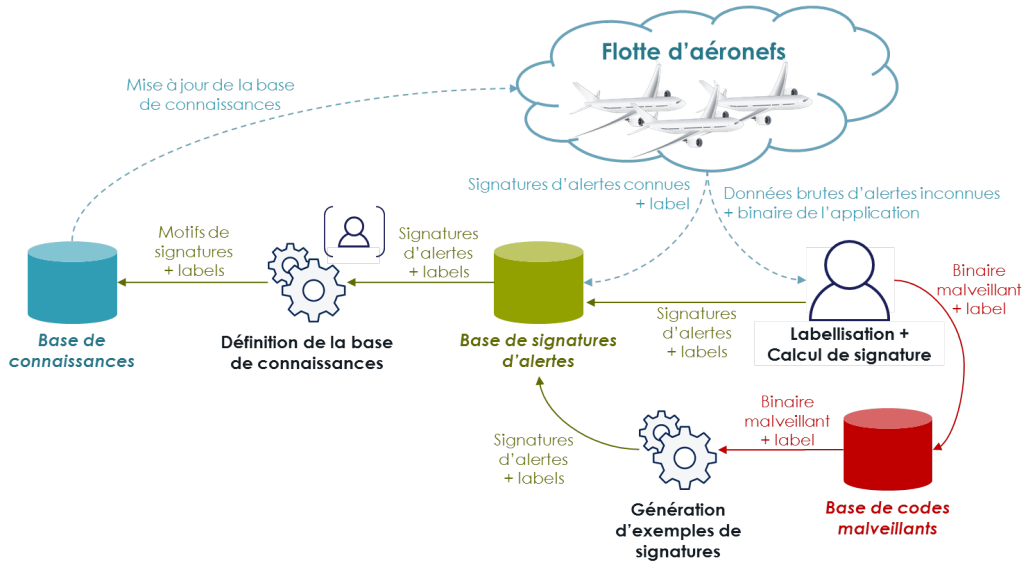


FIGURE 6.2 – Principe de la partie d'investigation au sol

6.1.4 Construction du message d'alerte

La dernière étape consiste à construire le message d'alerte. Ce message peut être construit différemment selon le label déterminé pour l'alerte. Si le label est connu, le message d'alerte peut par exemple n'être composé que du label (ou de l'ensemble des labels avec leur vraisemblance) et de la signature de l'alerte pour minimiser l'espace mémoire nécessaire. Si le label est inconnu, il est important de donner plus d'informations pour une investigation ultérieure au sol. Par exemple, un message relatif à une alerte *inconnue* peut également enregistrer les données brutes relatives à l'alerte et des données de contexte additionnelles.

Concernant le format des messages d'alerte, l'ARINC 852 (*Guidance for Security Event Logging in an IP Environment*) propose des recommandations quant au format des logs relatifs à la sécurité. Il serait donc intéressant d'utiliser ce format pour la construction des messages d'alerte de l'HIDS.

6.1.5 Investigation au sol

Le principe de la phase d'investigation au sol est détaillé par la Figure 6.2. Les messages d'alerte relevés par l'HIDS sur un ensemble d'aéronefs donné, et les données associées, sont utilisés comme entrées pour cette partie d'investigation au sol.

Les alertes reconnues par la confirmation d'attaque sont directement enregistrées dans la *Base de signatures d'alertes* avec le label associé. Dans le cas d'alertes inconnues, un expert est chargé d'effectuer un diagnostic relatif à cette alerte à l'aide des données associées à l'alerte et du binaire de l'application concernée. Cette phase de diagnostic doit lui permettre de définir le label relatif à l'alerte (attaque,

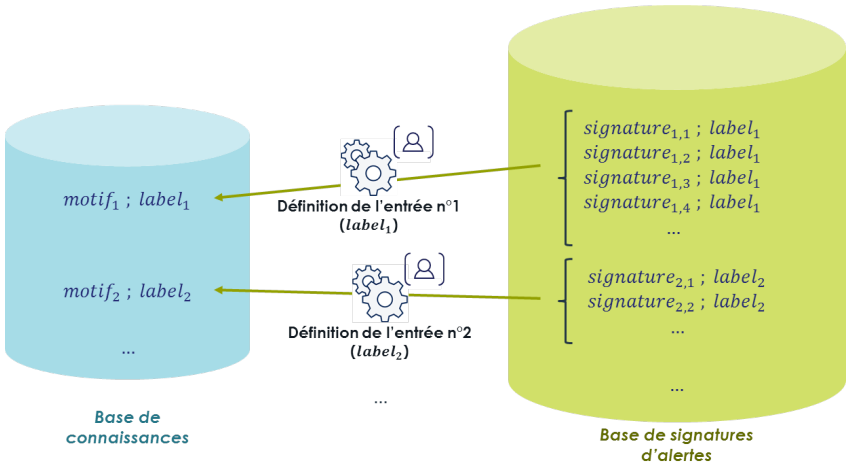


FIGURE 6.3 – Principe de la construction de la base de connaissances, à partir de la base de signatures d'alertes (au sol)

défaillance, ou fausse alerte). Il calcule ensuite la signature associée pour ajouter une nouvelle entrée dans la *Base de signatures d'alertes*. Dans le cas d'une application malveillante, le binaire correspondant et le label associé sont stockés dans une *Base de codes malveillants*. Cette base peut être utilisée afin de générer des exemples de signature pour un label donné, et ainsi enrichir la *Base de signatures d'alertes* directement au sol.

Enfin, les données de la *Base de signatures d'alertes* sont utilisées pour générer, de façon manuelle ou automatique, les entrées de la *Base de connaissances*. La Figure 6.3 schématise ce processus. Les entrées de la base de connaissances agrègent les signatures de la *Base de signatures d'alertes* afin de retrouver des motifs de signatures capables de représenter l'ensemble des signatures relevées pour un même label. Elles permettent ensuite de mettre à jour la base de connaissances embarquée dans la partition d'HIDS. Pour cette partie de mise à jour, il peut être envisageable de mettre à jour la base entière, ou seulement certaines entrées afin de gagner en efficacité.

Par principe, la base de signatures d'alertes est vide lors de la première mise en exploitation d'un aéronef, puisqu'aucune alerte n'a encore été levée. Le déploiement d'une base de connaissances vide n'est pas problématique dans notre approche, puisque les alertes relevées à bord seront donc toutes labellisées comme "inconnues" dans un premier temps. Cependant, il est possible que la base de codes malveillants soit non vide, ce qui peut permettre de générer des signatures d'alertes correspondantes au sol, et ainsi définir une base de connaissances avant la première mise en exploitation d'un aéronef. Également, les attaques utilisées pour la phase de validation du SDA peuvent servir à enrichir cette base de signatures d'alertes (et donc, la base de connaissances), si les attaques utilisées sont suffisamment représentatives d'attaques réelles. Enfin, si certains cas de fausses alertes ont été rencontrés au cours de l'intégration, ils peuvent également être enregistrés dans la base de signa-

tures d'alertes, et définir une ou plusieurs entrées "fausse alerte" dans la base de connaissances.

6.1.6 Conclusion

Le principe présenté ici cherche à donner des pistes quant à la consolidation d'un message d'alerte dans un contexte avionique. Le calcul d'un label correspondant à l'alerte en fonction d'une base de connaissances permet de préparer la phase de **Traitement du message d'alerte à bord**, relative à la réaction après alerte et qui n'est pas traitée dans ces travaux. L'introduction de la base de connaissances permet de donner une forte confiance dans les labels calculés et offre une bonne flexibilité pour opérer l'HIDS (seule la base de données est mise à jour, pas la partition d'HIDS en elle-même). La capacité à définir le label comme *inconnu* permet également de distinguer plusieurs niveaux d'information à attacher à l'alerte, afin d'optimiser l'espace mémoire de stockage d'un message d'alerte. Si l'alerte est connue, on peut envisager d'enregistrer uniquement la signature de l'alerte, tandis que si l'alerte est inconnue, il sera plus intéressant de conserver les données brutes ayant permis de calculer cette signature. Enfin, le rôle de l'expert au sol reste primordial pour donner un label aux nouvelles signatures d'alerte. Cependant, il est envisageable de définir de façon automatique les entrées de la base de connaissances en conservant au sol l'ensemble des signatures d'alerte relevées pour une flotte donnée. De plus, il est possible de conserver plusieurs niveaux de données au sol comme les signatures d'alertes et les codes malveillants rencontrés afin d'affiner la base de connaissances avec le temps d'une part, et donner les moyens de la pré-construire pour de futurs programmes avioniques d'autre part.

6.2 Implémentation

Le principe présenté dans la section précédente a été implémenté dans notre environnement d'étude. Cet environnement se compose ici d'un banc de test (lui-même composé d'une cible et d'un contrôleur), et d'une station de prototypage. L'architecture du prototype est décrite par la Figure 6.4. Sur la cible, on retrouve l'environnement de base composé du RTOS et d'une application avionique d'affichage, auxquels viennent s'ajouter la partition d'HIDS et le moniteur de SDA. Sur le contrôleur, deux outils sont utilisés afin de générer des comportements malveillants (*Injection d'attaque*) et de collecter des données pour la phase de prototypage (*Collecteur*). L'outil d'*Injection d'attaque* est utilisé pour modifier l'application d'affichage de façon malveillante, afin de provoquer la génération d'alertes par la partition d'HIDS. Ces alertes sont interceptées par le *Collecteur* afin d'enregistrer les informations brutes correspondantes. Enfin, on retrouve trois éléments sur la station de prototypage : l'*Extraction de la signature*, la *Construction de la base de connaissances*, et la *Recherche dans la base de connaissances*. Tout d'abord, les données extraites par le *Collecteur* sont utilisées sur la station de prototypage pour générer des signatures (*Extraction de la signature*). Ces signatures peuvent ensuite

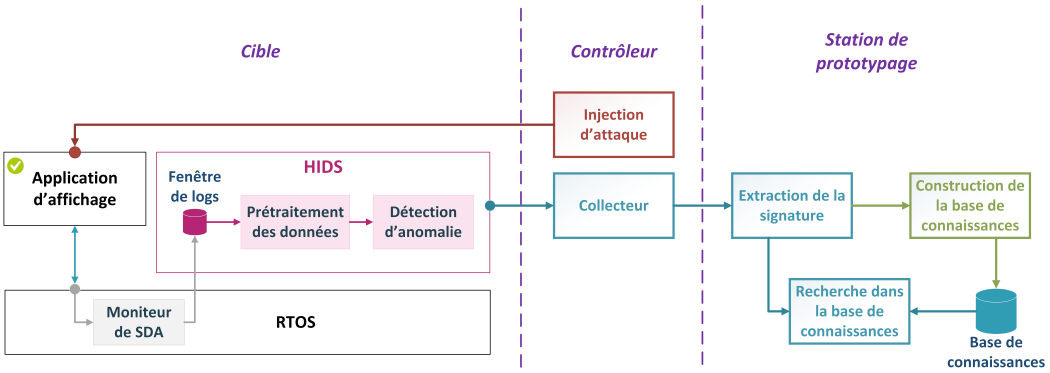


FIGURE 6.4 – Prototype implémenté pour l'aide au diagnostic

être utilisées de deux façons. Soit elles permettent de construire la base de connaissances (*Construction de la base de connaissances*), soit elles permettent de la tester une fois celle-ci construite (*Recherche dans la base de connaissances*).

Les parties suivantes présentent de façon plus détaillée l'implémentation de chaque composant.

6.2.1 Description de la cible

Pour ces travaux, la version de l'HIDS utilisée est la version **AT_comms** présentée au chapitre précédent. C'est également l'application IHM-DV qui est surveillée dans ce prototype. Les composants de *Moniteur de SDA* et *Partition HIDS* ont été légèrement modifiés afin d'enregistrer les informations nécessaires aux activités de *Confirmation d'attaque*.

6.2.1.1 Évolution du *Moniteur de SDA*

Le *Moniteur de SDA* est un code d'instrumentation inséré sur la cible afin d'intercepter les appels API ARINC 653 effectués par l'application IHM-DV. Ce moniteur est inséré dans une fonction permettant d'effectuer des appels système, qui est appelée par l'ensemble des appels API de l'application. Initialement, ce moniteur enregistrait uniquement l'ID de l'appel et son horodatage (Données d'exécution). Dans cette version, les adresses de retour de fonction consécutives sont également sauvegardées, sur quatre niveaux. Plus précisément le premier niveau correspond à l'instruction de l'appel API A ayant appelé le code commun d'appel système, le deuxième niveau correspond à l'instruction de la fonction F ayant effectué l'appel API A, le troisième niveau correspond à l'instruction de la fonction G ayant fait appel à la fonction F, et le quatrième niveau correspond à l'instruction dans la fonction H ayant fait appel à la fonction G. Ces informations se retrouvent facilement en inspectant les registres et la pile. Le registre LR correspond à l'adresse de retour courante (premier niveau). Les autres adresses se retrouvent à l'aide de la pile, qui est pointée par le registre $R1$. La Table 6.2 résume les six données d'exécution

TABLE 6.2 – Données d'exécution enregistrées par le *Moniteur de SDA*

Information	Registre correspondant	Description
ID de l'appel	R4	Identifiant unique de l'appel API
Horodatage	TBL	Valeur de l'horloge interne du RTOS
Adresse de retour (Niveau 1)	LR	Adresse de retour courante
Adresse de retour (Niveau 2)	*(*R1+4)	1ère adresse de retour enregistrée sur la pile (qui sera enregistrée dans le registre LR à la prochaine instruction RET)
Adresse de retour (Niveau 3)	*(**R1+4)	2ème adresse de retour enregistrée sur la pile
Adresse de retour (Niveau 4)	*(***R1+4)	3ème adresse de retour enregistrée sur la pile

enregistrées par le *Moniteur de SDA* à chaque appel API effectué par IHM-DV, en précisant la façon dont elles sont calculées dans le code d'instrumentation.

La taille de ce code d'instrumentation est de 212 octets (53 instructions), ce qui représente une augmentation de 72 octets par rapport à la version précédente.

6.2.1.2 Évolution de la *Partition HIDS*

La partition d'HIDS a été légèrement modifiée afin de conserver dans la *Fenêtre de logs*, pour chaque appel API, les six éléments décrits précédemment (ID de l'appel, horodatage, adresse de retour n°1, adresse de retour n°2, adresse de retour n°3, et adresse de retour n°4). Les parties de *Prétraitement des données* et de *Détection d'anomalie* restent identiques, et ne prennent donc en compte que les éléments "ID de l'appel" et "horodatage" des logs. Lorsqu'une anomalie est trouvée, l'ensemble des données brutes la composant (les six informations de chaque log composant la séquence anormale) sont conservées dans le buffer d'anomalies. Comme précédemment, l'alerte est confirmée lorsque ce buffer est rempli. La structure de la partition reste donc la même, mais le modèle a dû être mis à jour puisque le *Moniteur de SDA* a été légèrement modifié.

6.2.2 Injection d'attaque

Pour ces travaux, quatre attaques ont été définies manuellement. Elles visent soit l'intégrité, soit la disponibilité de l'application, en modifiant les données affichées ou en provoquant l'interruption de l'application. Ces attaques ont été réalisées manuellement en utilisant les opérateurs d'attaques définis dans l'outil d'injection. Elles ont été réalisées dans le but de provoquer une perte de confiance du pilote dans son affichage, sans connaissance préalable sur le fonctionnement de l'application. Finalement, ce sont des fonctions très utilisées qui ont été attaquées, ce qui a

TABLE 6.3 – Attaques réalisées pour les travaux d'aide au diagnostic

ID	Nom de l'attaque	Opérateur utilisé	Nombre de versions	Description
1	Modifie_NOM	MV	3	Remplace le nom d'un champ affiché à l'écran par la valeur "PWN"
2	Supprime_GT	SF	5	Supprime des appels à la fonction GET_TIME
3	Supprime_PW	SF	7	Supprime des appels à la fonction PERIODIC_WAIT
4	Supprime_RSM	SF	4	Supprime des appels à la fonction READ_SAMPLING_MESSAGE

provoqué des impacts sur l'intégrité ou la disponibilité de l'affichage de l'application.

Afin de définir plusieurs variantes pour chaque attaque, celles-ci ont été injectées à l'aide de l'outil d'injection d'attaque avec des paramétrages de contrôle de durée et de fréquence d'activation différents. La Table 6.3 liste ces quatre attaques développées avec les opérateurs d'attaque utilisés et une rapide description. Chaque classe d'attaque a été réalisée selon trois à sept paramétrages de contrôle d'activation de l'attaque différents. Ces variations de paramètres permettent de générer plusieurs versions d'une même attaque, qu'il serait intéressant de détecter de la même façon (leur donner le même label).

6.2.3 Collecteur

Le collecteur permet d'extraire les données brutes d'une alerte relevée par la partition d'HIDS (sur la cible) sous la forme d'un fichier de données (sur le contrôleur). Ces données sont traitées ultérieurement sur la station de prototypage.

Le collecteur est implémenté à l'aide d'un point d'arrêt GDB, qui se déclenche lorsque le buffer d'anomalies de la partition HIDS est plein. Lorsque le point d'arrêt est atteint, le script GDB enregistre dans un fichier une description de l'expérimentation courante (classe et version de l'attaque injectée, le cas échéant), et les données brutes de chaque anomalie dans le buffer (les six données [ID d'appel, horodatage, adresse de retour n°1, adresse de retour n°2, adresse de retour n°3, adresse de retour n°4] de chaque log composant une séquence anormale, pour chaque séquence anormale).

6.2.4 Extraction de la signature

Pour cette implémentation, 18 caractéristiques sont extraites des informations d'une alerte afin de générer une signature. Ces 18 caractéristiques sont détaillées dans la Table 6.4.

TABLE 6.4 – Caractéristiques utilisées comme signature d'une alerte

Nom	Description
$dist_{totale}$	Total des distances entre les anomalies et le modèle
$dist_{min}$	Distance minimale entre les anomalies et le modèle
$dist_{max}$	Distance maximale entre les anomalies et le modèle
$nb_{proches}$	Nombre d'anomalies très proches du modèle (dont la distance est inférieure à un certain seuil, placé à 20 tics d'horloge ici)
$nb_{inconnue}$	Nombre d'anomalies dont la séquence est inconnue
$nb_{ar,i}$	Nombre d'adresses de retour (ar) différentes de niveau i, $i \in [1, 2, 3, 4]$
nb_{ar_global}	Nombre d'adresses de retour uniques
$nb_{consecutives}$	Nombre maximal d'anomalies consécutives
$nb_{max\_par\_cycle}$	Nombre maximal d'anomalies observées sur un même cycle d'exécution
$nb_{groupes}$	Nombre de groupes d'anomalies ayant une suite d'adresses de retour identique
nb_i	Occurrences de l'appel API n°i dans l'ensemble des anomalies, $i \in \{42, 43, 47, 48, 50\}$ (IDs des appels API relatifs aux communications)

Les caractéristiques $dist_{totale}$, $dist_{min}$, $dist_{max}$, $nb_{proches}$, et $nb_{inconnue}$ se concentrent sur la distance calculée entre chaque anomalie formant l'alerte et le modèle SDA. Ici, cette distance se traduit par une valeur négative si la séquence n'existe pas dans le modèle (-1), ce qui permet de compter le nombre de séquences inconnues ($nb_{inconnue}$). Si la séquence existe, la distance représente la plus petite distance entre la durée observée pour la séquence, et les intervalles de durée autorisés pour cette séquence. Pour calculer le nombre d'anomalies proches du SDA ($nb_{proches}$), un seuil permet de définir si une anomalie est proche du modèle, en fonction de la distance au modèle calculée. La somme des distances, la plus petite distance, et la plus grande distance sont également conservées ($dist_{totale}$, $dist_{min}$, $dist_{max}$).

Les caractéristiques $nb_{ar,i}$, nb_{ar_global}, $nb_{groupes}$, nb_i permettent de représenter des liens entre les anomalies, notamment par rapport à leur origine (dans le code). Le nombre d'adresses de retours différentes observées parmi les anomalies (de façon locale avec $nb_{ar,i}$ ou globale avec nb_{ar_global}) et le nombre de groupes ($nb_{groupes}$) donnent plutôt des indications sur la variabilité des anomalies relevées (si elles se ressemblent ou non). Le nombre d'occurrences de l'appel API n°i (nb_i) indiquent plutôt les emplacements dans le code à l'origine des anomalies (appels ciblés).

Enfin, le nombre maximum d'anomalies consécutives ($nb_{consecutives}$) et le nombre maximum d'anomalies relevées sur un même cycle ($nb_{max\_par\_cycle}$) permettent de représenter l'aspect temporel d'apparition des anomalies.

Chacune de ces caractéristiques (et donc, la signature complète) est calculable directement depuis les données d'alerte et les données générales d'exécution qui y sont rattachées (les valeurs des adresses de retour). Les données de contexte n'ont pas été explorées dans ce prototype.

L'ensemble des signatures calculées à partir des données du *Collecteur* est ensuite stocké dans un fichier, dont une partie est utilisée pour remplir la base de connaissances. Le reste des exemples est utilisé pour tester l'exactitude de la recherche de signatures depuis la base de connaissances précédemment construite.

6.2.5 Construction de la base de connaissances

Trois implémentations ont été envisagées pour définir la base de connaissances et la construire, en fonction de la quantité d'exemples disponibles. Dans les trois cas, une entrée de la base de connaissances est composée d'une matrice 18×3 composée de 18 lignes représentant chacune une caractéristique, et 3 colonnes représentant la *valeur minimale acceptable*, la *valeur maximale acceptable*, et la *valeur médiane attendue* pour cette caractéristique.

La première implémentation consiste en une définition manuelle des règles de la base de connaissances par un expert, afin de généraliser un très faible nombre d'exemples disponibles (de l'ordre d'un ou deux exemples). Vu les caractéristiques utilisées, il semble peu judicieux d'enregistrer directement les signatures dans la base de connaissances, puisqu'il y a très peu de chances de retrouver la même signature (elle serait trop spécifique), et cela engendrerait une base de connaissances trop grande. Pour la généraliser, l'expert peut définir une entrée dans la base de connaissances comme 1) un intervalle possible pour chaque valeur de la signature, qui seront enregistrées comme *valeur minimale acceptable* et *valeur maximale acceptable*, et 2) la valeur de la signature originale enregistrée comme *valeur médiane attendue*.

Avec quelques exemples, on peut envisager une seconde implémentation plus automatisée qui consiste à construire un intervalle en fonction des valeurs observées pour une classe donnée, et à conserver la valeur médiane calculée sur ces exemples.

La dernière implémentation proposée est développée en python selon le pseudo-code décrit par le Listing 6.1. Celle-ci est envisageable avec un nombre suffisant d'exemples d'une même classe d'attaque, et est intéressante dans le cas où plusieurs versions d'une même attaque exhibent des comportements différents. Pour ce troisième cas, l'algorithme effectue une boucle pour traiter indépendamment les signatures correspondant à chaque label (ligne 8). Tout d'abord, seules les signatures correspondant au label courant sont sélectionnées (ligne 10), puis l'algorithme de clustering *k-means* est appliqué pour définir n_{label} groupes parmi ces signatures (ligne 12), n_{label} ayant été défini au préalable pour chaque label, indépendamment (ligne 4). Une entrée de la base de connaissances est ensuite créée pour chaque groupe (lignes 14 à 16). Ces entrées ont le même format que pour les deux implémentations précédentes.

Il est donc possible de faire évoluer la façon de définir les entrées de la base de

```
# Chargement des exemples de signatures labellisés
data = charger("base_de_signatures.csv")
# Choix du nombre de sous-classe, pour chaque classe
nb_groupes = choisir_nombre_de_groupes_par_label(data)
# Initialisation de la base de connaissances
base_de_connaissances = dict()
# Parcours de toutes les classes pour créer une ou plusieurs entrées correspondantes dans la base de connaissances
for (i,label) in enumerate(data.labels) :
    # Utilisation des signatures correspondant à la classe courante
    signatures = data["label"==label]
    # Groupement des signatures en à l'aide de l'algorithme de clustering k-means
    groupes = kmeans(signatures,nb_groupes[i])
    # Création d'une entrée dans las base de connaissances, pour chaque groupe de signatures
    for groupe in groupes :
        entree = creer_entree(groupe)
        base_de_connaissances[entree.label] = entree.motif
return base_de_connaissances
```

Listing 6.1 – Définition automatique de la base de connaissances (Pseudo-code)

connaissances en fonction de la quantité d'exemples disponibles, tout en gardant une définition commune de ces entrées. Cela permet une certaine flexibilité dans la définition de la base de connaissances sans modifier le code qui sera embarqué pour effectuer cette vérification.

6.2.6 Vérification de la base de connaissances

Pour vérifier si une signature est dans la base de connaissances, on regarde si les valeurs de chaque caractéristique sont contenues dans l'intervalle d'une entrée de la base, pour chaque entrée. Plusieurs cas sont possibles :

— Si aucune entrée ne correspond, alors le label renvoyé est "inconnu".

— Si une seule entrée correspond, alors le label renvoyé est celui de cette entrée.

— Si plusieurs entrées correspondent, alors un calcul de distance est effectué entre la signature et la valeur médiane de chaque entrée. Le label final est une concaténation de l'ensemble des labels, avec la distance calculée correspondante. Dans l'implémentation proposée ici, une distance euclidienne a été utilisée pour déterminer la distance entre la signature et la médiane d'une entrée de la base de connaissances.

Il est important de prendre en compte le cas où plusieurs entrées peuvent correspondre à la signature, puisque la construction des intervalles peut engendrer des recoupements entre plusieurs entrées de la base de connaissances. Le choix a

été fait ici de donner l'ensemble des labels correspondants en les classant par distance, plutôt que de ne renvoyer que le label le plus probable. C'est la stratégie qui est également adoptée pour la fonction *BITE* par exemple, pour les messages de maintenance.

6.3 Expérimentations

Les expérimentations présentées ici exposent les premiers résultats obtenus à l'aide du prototype précédemment développé. La première partie présente les données utilisées pour ces expérimentations. Ensuite, cette section présente une première évaluation des caractéristiques sélectionnées afin de comprendre si elles permettent de répondre aux objectifs énoncés dans la Section 6.1.2. Les résultats ayant été plutôt positifs, les deux parties suivantes présentent 1) une expérimentation réalisée pour tester la définition automatique des signatures et 2) une évaluation des ressources nécessaires pour exécuter la partie de confirmation d'attaque à bord.

6.3.1 Jeux de données

Le jeu de données utilisé dans ces expérimentations est composé de 1208 alertes envoyées par la *Détection d'anomalie*. Elles sont réparties de la façon suivante :

— 246 fausses alertes (alertes générées pendant une exécution normale),

— 214 attaques "Modifie_NOM" (n°1),

— 270 attaques "Supprime_GT" (n°2),

— 219 attaques "Supprime_PW" (n°3), et

— 259 attaques "Supprime_RSM" (n°4).

Les alertes relatives aux attaques ont été générées en activant l'outil d'injection d'attaque selon les attaques décrites précédemment dans la Table 6.3.

Pour générer les fausses alertes, la partition d'HIDS a été légèrement modifiée afin de supprimer la période de validité des anomalies. Ainsi, les anomalies trop anciennes ne sont plus supprimées du buffer d'anomalies. Par conséquent, une alerte est levée dès que p anomalies ont été relevées, depuis le début de l'exécution de la partition IHM-DV. Pour les expérimentations réalisées dans ce chapitre, la calibration du SDA nous a mené à fixer $p = 10$.

Dans les deux cas (exécution normale ou contenant une attaque), une fois l'alerte enregistrée, le buffer d'anomalies de la partition HIDS est remis à zéro afin de collecter plusieurs exemples d'alerte pendant une même exécution. L'ensemble des alertes enregistrées pour une même classe sont toutefois issues de plusieurs exécutions. Du fait du manque de mécanismes de tolérance aux fautes présent sur notre cible (en cours de développement), aucun exemple de défaillance n'a été évalué ici.

6.3.2 Choix des caractéristiques

Cette première expérimentation cherche à évaluer la pertinence des caractéristiques choisies pour décrire la signature d'une alerte. Le but est donc de vérifier la capacité à séparer correctement le jeu de données selon le label associé. L'outil Weka [1] propose un module d'exploration de données qui a été utilisé ici pour explorer les données d'alerte de notre jeu de données.

Dans un premier temps, l'analyse en composantes principales ou PCA (*Principal Components Analysis*) des données a montré une forte corrélation (entre 89% et 98%) entre les caractéristiques $nb_{ar,2}$, $nb_{ar,3}$, $nb_{ar,4}$, nb_{ar_global}, et $nb_{groupes}$, quel que soit le type de signature (classe d'attaque ou fausse alerte). Il n'est donc pas forcément nécessaire de conserver l'ensemble de ces caractéristiques dans une signature, ce qui permettrait de simplifier celle-ci.

Ensuite, l'algorithme de clustering k-means (apprentissage non supervisé) a été utilisé pour séparer les données d'alerte en 5 clusters, soit autant de clusters que de classes attendues (la classe de fausse alerte et les quatre classes d'attaque). En utilisant l'ensemble des 18 caractéristiques, l'algorithme confond les deux classes d'attaque *Supprime_GT* et *Supprime_RSM* sur plusieurs instances, ce qui résulte en un total de 211 instances mal classées, soit 17.47% des instances. En supprimant les caractéristiques $nb_{ar,2}$, $nb_{ar,3}$, $nb_{ar,4}$ et nb_{ar_global}, qui sont fortement corrélées avec la caractéristique $nb_{groupes}$, l'algorithme donne de bien meilleurs résultats puisque seules 56 instances sont mal classées, ce qui correspond à 4.64% des instances. Les caractéristiques proposées semblent donc suffisamment pertinentes pour différencier les cinq classes étudiées.

Un arbre de décision (apprentissage supervisé) a également été utilisé afin de classer les données d'alerte de façon supervisée. L'algorithme a permis d'atteindre un taux de 95.94% d'instances correctement classées d'une part, mais également de mettre en évidence l'importance des caractéristiques sur la distance ($dist_{totale}$, $dist_{min}$, $dist_{max}$, $nb_{proches}$) et sur les appels API impactés ($nb_i, i \in 42, 43, 47, 48, 50$). En effet, l'arbre de décision final se base essentiellement sur ces caractéristiques pour effectuer son classement.

Cette première expérimentation a donc permis de montrer l'intérêt des caractéristiques choisies, et a permis d'en sélectionner le sous-ensemble le plus pertinent. Dans la suite des expérimentations, les 4 caractéristiques $nb_{ar,2}$, $nb_{ar,3}$, $nb_{ar,4}$, nb_{ar_global} ne seront donc plus utilisées (les caractéristiques utilisées seront donc au nombre de 14, résumées dans la Table 6.5). Cependant, cette expérimentation montre également qu'il est difficile de définir une signature sur les données d'alerte qui soit parfaitement efficace (100% d'instances bien classées). En effet, certaines données sont très proches voire égales, sans pour autant appartenir à la même classe. Il serait intéressant d'étudier d'autres données telles que les données de contexte pour parvenir à différencier ces cas particuliers.

1. https ://www.cs.waikato.ac.nz/ml/weka/

TABLE 6.5 – Caractéristiques retenues pour définir la signature d'une alerte

Nom	Description
$dist_{totale}$	Total des distances entre les anomalies et le modèle
$dist_{min}$	Distance minimale entre les anomalies et le modèle
$dist_{max}$	Distance maximale entre les anomalies et le modèle
$nb_{proches}$	Nombre d'anomalies très proches du modèle (dont la distance est inférieure à un certain seuil, placé à 20 tics d'horloge ici)
$nb_{inconnue}$	Nombre d'anomalies dont la séquence est inconnue
$nb_{ar,1}$	Nombre d'adresses de retour différentes de niveau 1
$nb_{consecutives}$	Nombre maximal d'anomalies consécutives
$nb_{max\_par\_cycle}$	Nombre maximal d'anomalies observées sur un même cycle d'exécution
$nb_{groupes}$	Nombre de groupes d'anomalies ayant une suite d'adresses de retour identique
nb_i	Occurrences de l'appel API n°i dans l'ensemble des anomalies, $i \in 42, 43, 47, 48, 50$ (IDs des appels API relatifs aux communications)

6.3.3 Définition automatique de la base de connaissances

L'expérimentation présentée ici cherche à évaluer la dernière implémentation proposée pour définir automatiquement les entrées de la base de connaissances à partir d'exemples de signatures. Pour cette implémentation, les signatures d'alerte sont tout d'abord regroupées par classe (fausse alerte ou classe d'attaque spécifique). Ensuite, pour chaque classe, une ou plusieurs entrée(s) de la base de connaissances sont créées automatiquement en sélectionnant, pour chaque caractéristique, la valeur minimale, la valeur maximale, et la valeur médiane observée dans le groupe correspondant à la classe.

Dans un premier temps, il est nécessaire de définir le nombre de groupes (ou sous-classes) à considérer pour chaque classe d'attaque. Un première étude de l'ensemble des exemples de signatures a permis de mettre en évidence le fait que les classes d'attaque "Modifie_NOM" et "Supprime_GT" présentent deux sous-classes distinctes. Ces sous-classes dépendent directement des paramètres utilisés pour définir des variantes aux attaques (contrôle du moment et de la fréquence d'activation de l'attaque). Pour l'attaque "Modifie_NOM", on distingue deux sous-classes en fonction de la durée d'activation choisie (dès la première exécution de l'application ou avec un départ différé). Pour l'attaque "Supprime_GT", on distingue également deux sous-classes, en fonction de la fréquence d'activation de la charge malveillante (activation permanente ou intermittente). Par conséquent, le nombre de groupes a été fixé à 2 pour ces deux attaques ("Modifie_NOM" et "Supprime_GT"), tandis que le nombre de groupes a été fixé à 1 pour les trois autres classes ("Fausse alerte",

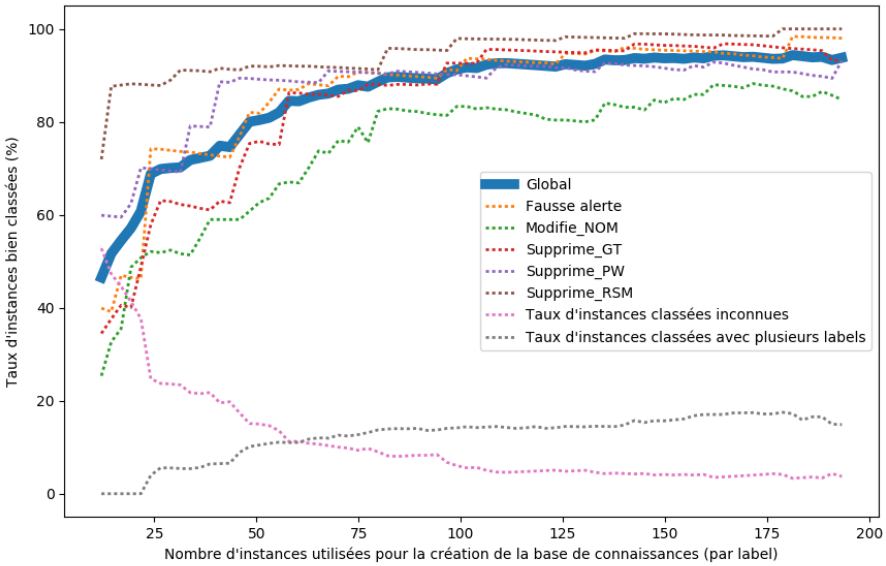


FIGURE 6.5 – Taux d'instances classées correctement, en fonction de la quantité de données utilisée pour l'entraînement

"Supprime_PW", et "Supprime_RSM"). Le nombre total d'entrées dans la base de connaissances est donc fixé à 7.

La Figure 6.5 présente le taux d'instances bien classées, en fonction du nombre d'instances utilisées pour construire la base de connaissances. Le taux d'instances bien classées est calculé sur les instances qui n'ont pas été utilisées pour l'apprentissage (instances de test). Une instance est considérée comme bien classée si le label donné est celui attendu. Si plusieurs labels sont donnés, une instance est considérée comme bien classée si le bon label fait partie des labels donnés. Le taux d'instances inconnues et le taux d'instances classées avec plusieurs labels est également indiqué sur la figure.

Globalement, le taux d'instances bien classées s'améliore avec le nombre de données utilisées pour construire la base de connaissances. Ce taux dépasse les 90% lorsque plus de 40% des instances sont utilisées pour la construction de la base de connaissances, ce qui correspond à une centaine d'instances pour chaque classe. A ce point, on observe également 6.76% d'instances classées comme inconnues, et 14.07% d'instances ayant plusieurs labels. La forme asymptotique de la courbe représentant le taux global d'instances bien classées indique également que cette quantité de données (une centaine d'exemples de chaque classe) est suffisante pour définir des entrées de base de connaissance suffisamment précises.

On observe également des différences selon les classes. Par exemple, la classe "Supprimer_RSM" est très bien repérée (taux d'instances bien classées supérieur à 90% à partir de 13% d'instances utilisées pour la construction de la base de signa-

tures, soit environ 32 instances), tandis que la classe "Modifie_NOM" est la plus difficile à classer correctement. Cette différence peut s'expliquer par le choix des données utilisées pour détecter des anomalies. L'introduction de données relatives à l'état de la mémoire pourrait permettre de mieux distinguer les effets de l'attaque "Modifie_NOM", qui a un impact direct sur la mémoire, et donc de mieux la caractériser.

Enfin, on peut voir que le taux d'instances classées avec plusieurs labels est assez conséquent. Celui-ci atteint environ 15% lorsque plus de 100 exemples de signatures de chaque classe sont utilisés pour l'apprentissage. Pour cette expérimentation, dans le cas où une signature correspond à plusieurs labels, un calcul de distance est effectué entre la signature et la valeur médiane enregistrée dans l'entrée de la base de connaissances correspondant à ce label, pour chaque label. Cependant, ce calcul de distance a été implémenté sous la forme d'une distance euclidienne qui n'est pas très adaptée au problème formulé ici. Ces distances calculées n'ont donc pas été utilisées pour discriminer les labels entre eux (e.g. donner le label le plus probable), mais seulement à titre indicatif. Une amélioration de l'implémentation proposée ici consisterait à définir une stratégie plus efficace pour sélectionner le label le plus probable, lorsqu'une signature correspond à plusieurs entrées de la base de connaissances.

Pour conclure, si la quantité d'exemples nécessaires dans l'implémentation proposée reste relativement importante, elle peut être rapidement atteinte si 1) plusieurs alertes sont générées pour une même attaque, ou si 2) il est possible d'exécuter l'attaque sur un banc de test et ainsi générer la quantité d'exemples nécessaire. L'utilisation d'une autre représentation des entrées de la base de connaissance, par exemple en se basant sur des modèles statistiques tels que les réseaux bayésiens naïfs, peut également permettre d'optimiser le nombre d'exemples nécessaire pour définir une entrée efficace dans la base de connaissances.

Les résultats obtenus sont donc encourageants quant à la possibilité de construire une base de connaissances de façon automatique à partir d'exemples de signature étiquetés. Afin d'estimer la consommation de ressources réelle associée à cette solution d'aide au diagnostic, celle-ci a été implémentée dans la *Partition HIDS*. Les résultats associés sont présentés dans la section suivante.

6.3.4 Utilisation des ressources

Cette dernière partie présente l'évaluation en terme de ressources de la solution d'aide au diagnostic proposée ici. Dans un premier temps, l'impact du *Moniteur de SDA* est réévalué, puisque cette fois le moniteur doit enregistrer plus d'informations. Dans un second temps, le principe de confirmation d'attaque a été implémenté dans une partition avionique afin d'évaluer la taille totale de la partition d'HIDS (Détection d'anomalie + Confirmation d'attaque). Finalement, cette implémentation embarquée a été utilisée pour évaluer le temps de calcul nécessaire à la construction et à la vérification d'une signature.

6.3.4.1 Temps de calcul pour le *Moniteur de SDA*

De la même façon que dans la Section 5.4.1, deux types de captures sont réalisées afin d'évaluer l'impact en terme de temps de calcul introduit par le *Moniteur de SDA*. Chaque capture cherche à enregistrer la durée d'exécution des appels système lorsque 1) le *Moniteur de SDA* est activé et 2) le *Moniteur de SDA* n'est pas activé. Plus de 11000 captures ont été réalisées pour chaque expérimentation.

La Figure 6.6a donne la distribution des durées observées, par type d'appel API, avec ou sans instrumentation. La Figure 6.6b présente cette distribution en regroupant les durées observées pour les appels API de communication d'une part, et les autres appels d'autre part.

Sur la première figure, on peut voir un impact net de l'instrumentation sur les appels API relatifs aux communications. De manière générale, cet impact peut être caractérisé par la différence au niveau de la médiane observée sur la deuxième figure, qui correspond à 6 ticks d'horloge, soit une augmentation d'environ 3.3%. Cette augmentation reste dans les objectifs fixés au Chapitre 1 en terme de consommation de ressources (moins de 5%).

Concernant les autres appels API, on observe un impact négatif de l'instrumentation (-1 tic d'horloge à la médiane). Ce résultat montre que l'impact de l'instrumentation n'est pas distinguable du bruit relatif à la durée d'exécution des appels API, pour les autres appels API. En effet, seules quelques instructions supplémentaires sont exécutées (12 instructions).

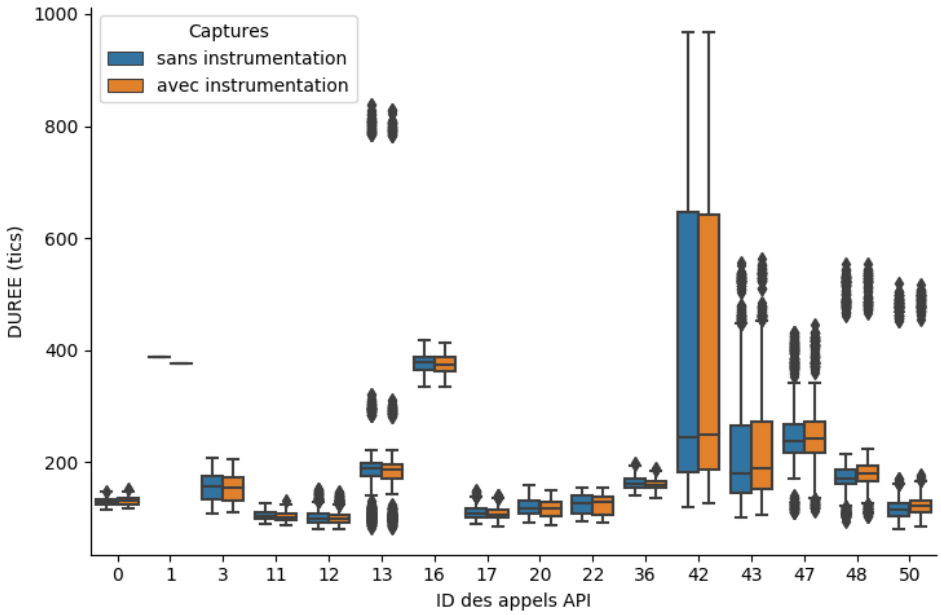
6.3.4.2 Espace mémoire pour la *Partition HIDS*

La Figure 6.7 présente l'implémentation de la *Partition HIDS* prenant en compte la confirmation d'attaque. Les parties d'**Extraction de la signature**, de **Recherche du label** et de **Construction du message d'alerte** ont été ajoutées à la suite de la partition d'HIDS proposée dans le chapitre précédent, pour la solution **AT_comms**. Ces trois étapes ne sont exécutées que si suffisamment d'anomalies ont été relevées par la **Détection d'anomalies**, sur une fenêtre de temps donnée.

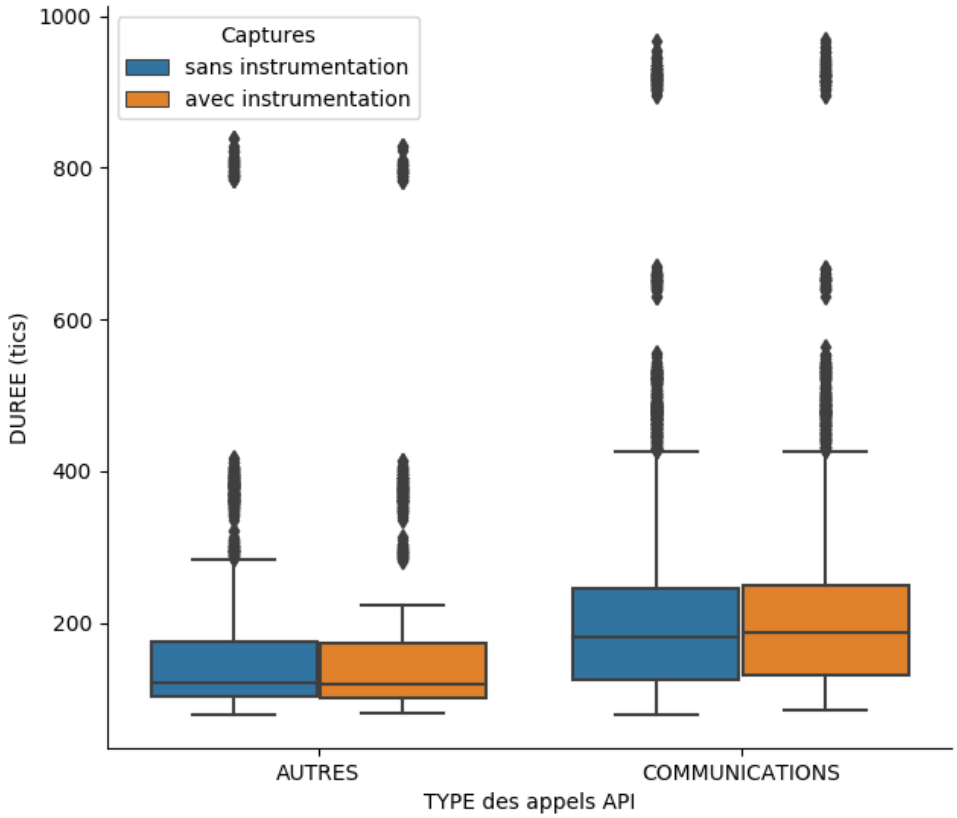
Le pseudo-code donné par le Listing 6.2 décrit la structure du code de la partition HIDS finale, avec les trois étapes relatives à la confirmation d'attaque. Une fois compilé, le code de la partition HIDS est de 48.8ko, ce qui représente 0.18% de la taille de la partition surveillée IHM-DV. Cela représente également une augmentation de 20.4ko par rapport à la version d'HIDS du chapitre précédent (sans confirmation d'attaque). Malgré une augmentation non négligeable de la taille de la partition, celle-ci reste largement raisonnable comparé à la taille de la partition surveillée.

6.3.4.3 Temps de calcul pour la *Partition HIDS*

Deux types d'expérimentations ont été réalisées afin d'estimer la durée d'exécution de la partition d'HIDS intégrant la partie de confirmation d'attaque.



(a) Distribution par type d'appel API



(b) Corrélation des distributions pour les appels API de communication et pour les autres

FIGURE 6.6 – Distribution de temps d'exécution, avec ou sans instrumentation

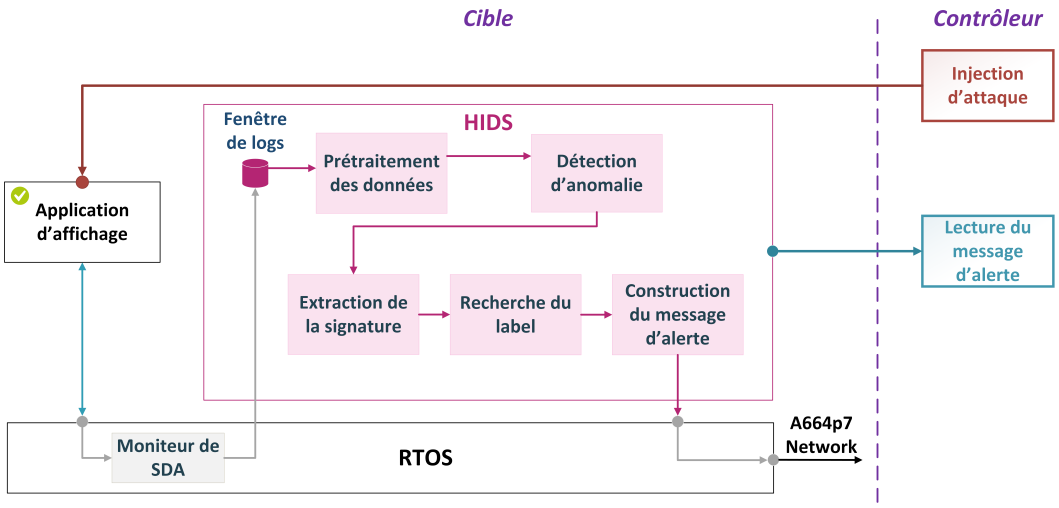


FIGURE 6.7 – Architecture de l'HIDS implémenté sur cible (Détection d'anomalie + Confirmation d'attaque)

Pour la première expérimentation, plusieurs captures de temps ont été effectuées afin de déterminer la durée d'exécution totale de la partition HIDS. Au cours de ces captures, deux modes d'exécution distincts ont été exécutés :

— Aucune alerte n'est relevée : exécution de la partie de *Détection d'anomalie* uniquement,

— Une alerte est relevée : exécution de la partie de *Détection d'anomalie* puis de la partie de *Confirmation d'attaque*.

La distribution de plus de 3500 captures de durée correspondantes est reportée sur la Figure 6.8, en fonction du nombre de logs traités par la partition HIDS. Le nombre de logs traité est également représenté par un dégradé de couleur. Ces captures de durée ont été effectuées en injectant ou non différentes attaques, qui peuvent faire varier le nombre de logs à traiter à chaque cycle (ici, entre 111 et 630 logs par cycle ont été traités).

Dans ces captures de durée, on voit apparaître sur la Figure 6.8 deux zones de durées pour un nombre de logs donné. Ces deux zones correspondent à la durée d'exécution de la partition HIDS quand la confirmation d'attaque est activée (zone supérieure, en cas d'alerte), ou quand elle n'est pas activée (zone inférieure, pas d'alerte). On peut également noter que le nombre de logs relevé ici (en cas d'attaque et sur plusieurs exemples différents) est assez différent du nombre de logs relevé en cas d'exécution normale (tel que capturé au chapitre précédent).

La deuxième expérimentation cherche à caractériser plus précisément la durée d'exécution du code de confirmation d'attaque. Pour cette expérimentation, environ 500 captures de durée ont été effectuées spécifiquement autour du code correspondant à la confirmation d'attaque. La distribution correspondante est présentée par la Figure 6.9. Sur cette figure, on distingue les distributions de durée en fonction du

```
# Phase d'initialisation de la partition
init();
# Boucle infinie (traitement périodique)
while(1) {
    # Prétraitement des logs
    dp = pretraitement_des_données(fenetre_de_logs);
    # Détection d'anomalie
    anomalies += detection_d_anomalies(dp);
    # Code de confirmation d'attaque
    si (anomalies.est_plein) {
        signature = extraction_signature(anomalies);
        labels = recherche(anomalies,base_de_connaissances);
        message = construction_message_d_alerte(labels);
    }
    # Suppression des anomalies trop anciennes
    mise_a_jour(anomalies);
    # Attendre le prochain cycle d'exécution
    PERIODIC_WAIT();
}
```

Listing 6.2 – Structure de la partition HIDS avec détection d'anomalie et confirmation d'attaque (Pseudo-code)

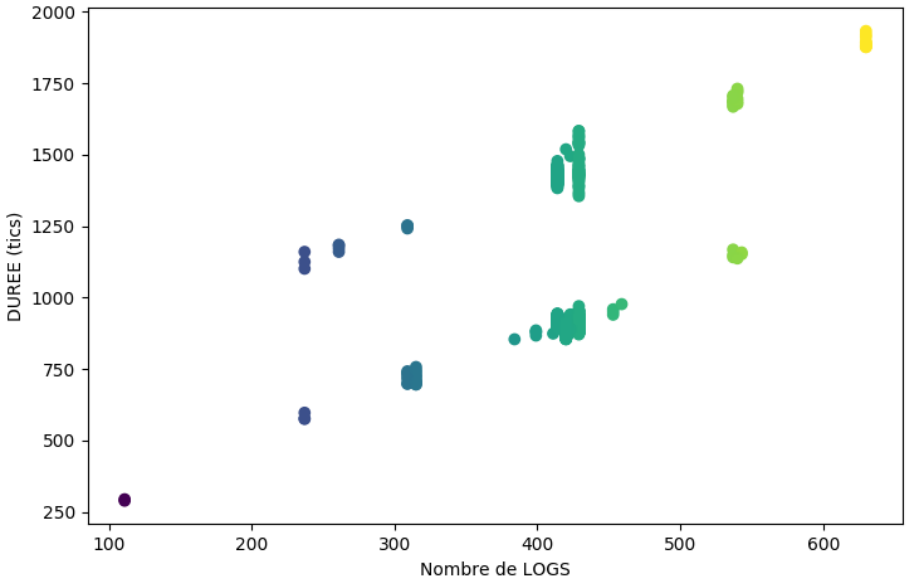


FIGURE 6.8 – Durée d'exécution de la partition d'HIDS, en fonction du nombre de logs à traiter

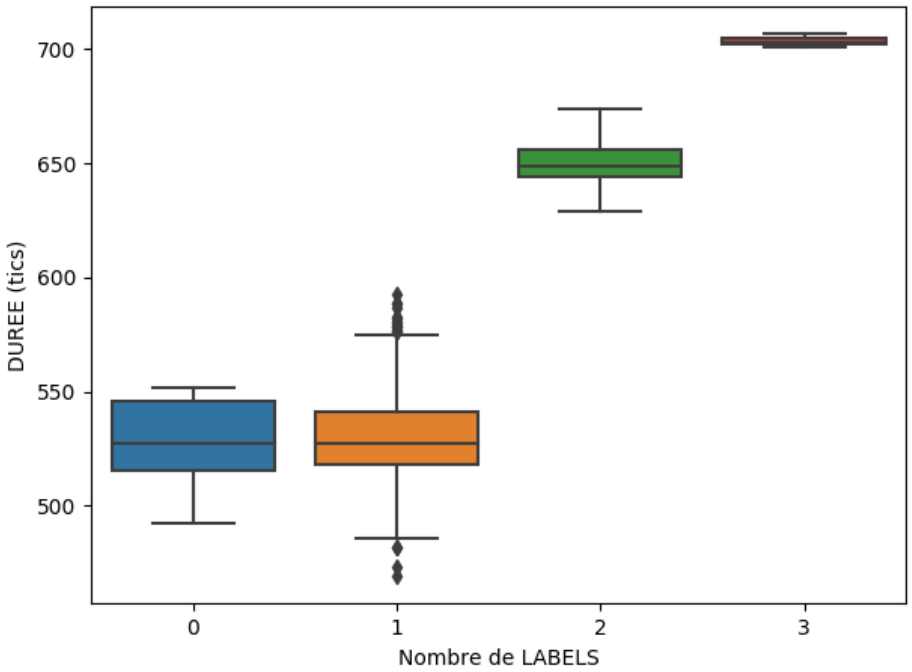


FIGURE 6.9 – Durée d'exécution de la Confirmation d'attaque, en fonction du nombre de labels correspondant à la signature

nombre d'entrées de la base de connaissances correspondant à la signature (Nombre de labels).

En effet, si le nombre de labels est de 0 ou 1, seule la comparaison de la signature avec les bornes inférieure et supérieure de chaque entrée de la base de connaissance est effectuée. Si le nombre de labels est strictement supérieur à 1, un calcul de distance entre la signature et la valeur médiane de chaque entrée correspondante de la base de connaissance est effectué, ce qui augmente la durée d'exécution de la confirmation d'attaque proportionnellement au nombre de labels correspondants à la signature évaluée.

En pratique, nous avons observé une durée médiane de 527 ticks dans le premier cas (nombre de labels à 0 ou 1). Pour le second cas, nous avons observé respectivement une durée médiane de 649 ticks et de 703 ticks pour le traitement d'une signature correspondant à 2 et à 3 labels. Ces valeurs sont bien cohérentes avec la différence observée entre les deux zones de la Figure 6.8.

De plus, parmi les captures effectuées, seules trois signatures correspondaient à trois entrées de la base de connaissances, et aucune signature ne correspondait à plus de trois entrées. En effet, la base de connaissance ne contient ici que sept entrées, ce qui limite les possibilités de chevauchement des motifs de signature, et donc les possibilités qu'une même signature corresponde à plusieurs labels. Également, il

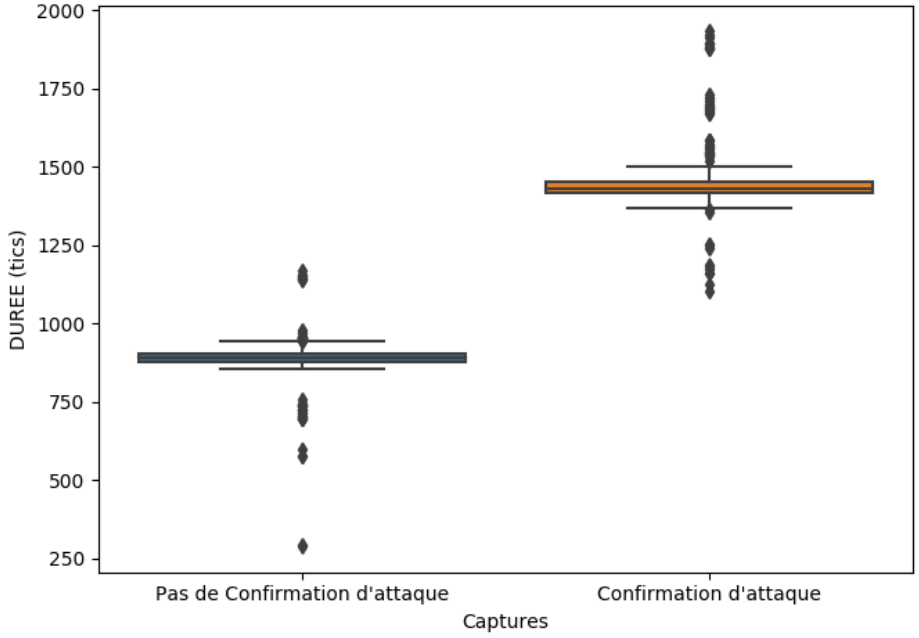


FIGURE 6.10 – Durée d'exécution de la partition d'HIDS, en fonction de l'activation ou non de la partie de Confirmation d'attaque

nous paraît vraisemblable dans un contexte industriel que la base de connaissance ne contienne que peu d'entrées, étant donné les difficultés actuelles pour attaquer un système avionique.

Finalement, les durées d'exécution capturées lors de la première expérimentation (durée d'exécution de la partition HIDS complète) ont été regroupées en fonction des deux zones observées (exécution ou non de la partie de confirmation d'attaque). Les distributions correspondantes sont données sur la Figure 6.10. En considérant les valeurs médianes observées (pour un nombre de logs traité compris entre 111 et 630), la partition d'HIDS nécessite entre 0.032ms (889 ticks, détection d'anomalie seule) et 0.051ms (1431 ticks, détection d'anomalie + confirmation d'attaque) pour s'exécuter. Ces durées représentent entre 0.47% et 0.76% du temps d'exécution alloué à la partition surveillée IHM-DV. Malgré une nette augmentation de la durée d'exécution, due à l'introduction de la *Confirmation d'attaque* dans la partition HIDS, celle-ci reste suffisamment faible au regard des objectifs énoncés (utilisation de moins de 5% des ressources allouées).

Néanmoins, aucune estimation du temps d'exécution dans le pire cas n'a été menée pour la partie de *Confirmation d'attaque*. Dans les expérimentations des chapitres précédents, nous avons montré que la partie de *Détection d'anomalie* de la solution **AT_comms**, reprise ici, pouvait souffrir d'un trop long temps de calcul si l'application surveillée utilise un grand nombre d'appels API (et notamment,

effectue des appels API en boucle). Sur la Figure 6.8, on observe d'ailleurs une augmentation linéaire du temps d'exécution de la partition HIDS, en fonction du nombre de logs à traiter. Ce problème peut être contourné en fixant une limite sur le nombre d'appels API réalisables par cycle d'exécution. Dans ce cas, si l'application réalise trop d'appels API pendant son cycle d'exécution, cela peut être considéré comme une anomalie. Concernant l'estimation du temps d'exécution dans le pire cas pour la partie de *Confirmation d'attaque*, on peut voir sur la Figure 6.9 que le temps d'exécution de cette partie dépend directement du nombre d'entrées de la base de connaissance correspondant à la signature d'alerte évaluée (Nombre de labels). En effet, un calcul supplémentaire (calcul de distance) est effectué si au moins deux labels peuvent correspondre à la signature évaluée. La durée d'exécution de cette partie du code est linéairement proportionnel au nombre de labels possibles (il y a un calcul de distance par label). L'estimation du pire temps d'exécution dépend donc de la base de connaissances, et peut être calculé an fonction du nombre maximum d'entrées à l'intérieur de la base qui peuvent se recouper. Ce calcul peut être effectué pendant la phase d'intégration, et permettre de dimensionner un seuil sur le nombre maximal d'entrées de la base de connaissances pouvant se recouper. La connaissance de ce seuil permettra par la suite de mettre à jour la base de connaissance tout en respectant un pire temps d'exécution donné, tout au long du cycle de vie de l'aéronef.

6.4 Conclusion

Ce chapitre a présenté quelques perspectives quant au développement des parties de *Confirmation d'attaque* et d'*Investigation au sol* de l'approche générale d'HIDS. En particulier, ce chapitre a présenté un principe de vérification des alertes levées par la *Détection d'anomalie* à l'aide d'une base de connaissances. Cette vérification permet de sélectionner un niveau d'information à envoyer avec l'alerte, afin d'aider le traitement de celle-ci à bord (par exemple, par un enregistrement, une sanction sur l'application, ou une transmission au pilote), mais aussi d'aider un expert à effectuer un diagnostic plus approfondi de l'alerte au sol. Dans ce cadre, plusieurs caractéristiques ont été proposées pour développer la signature d'une alerte et la comparer ensuite avec la base de connaissances. Des méthodes de construction de la base de connaissances ont aussi été proposées, et ont été testées sur un cas d'étude concret. Cette expérimentation a donné des résultats intéressants en terme de reconnaissance des signatures par la base de connaissances proposée. Notamment, l'implémentation proposée est capable de caractériser correctement plus de 90% des instances d'alertes étudiées à partir d'environ 100 exemples de signatures pour les cinq classes étudiées (fausse alerte et quatre classes d'attaque différentes). Ce résultat n'est encore pas suffisamment précis pour que cette solution soit embarquée dans une cible avionique, mais permet de donner des pistes de réflexion afin d'améliorer ces résultats. Le choix de caractéristiques supplémentaires (par exemple, des données relatives au contexte de l'application), ou l'observation d'autres données pour lever une alerte sont donc des pistes qu'il serait intéressant d'explorer dans la

suite de ces travaux.

En outre, cette dernière phase de l'approche générale d'HIDS a été implémentée sur une cible avionique afin d'évaluer sa consommation de ressources dans un cas d'utilisation général. Tout comme la version présentée au chapitre précédent (partition HIDS ne contenant que la partie de *Détection d'anomalie*), les conclusions à ce sujet sont très prometteuses puisque l'HIDS n'engendre pas de surconsommation de ressources excessive (moins de 5% des ressources allouées à la partition surveillée), et est donc capable de s'exécuter en temps réel sur le calculateur avionique. Cependant, aucune estimation de durée dans un pire cas d'exécution n'a été réalisée ici. Il serait également intéressant d'estimer la durée d'exécution en fonction de la taille de la base de connaissances, qui ne comporte ici que 7 entrées.

Concernant la partie d'*Investigation au sol*, seules des pistes concernant les données à enregistrer pour aider au diagnostic d'une alerte inconnue au sol ont été évoquées. Il serait intéressant d'étudier plus précisément la façon de mener un diagnostic après alerte dans un contexte avionique, au regard des données de journalisation d'alerte proposées dans ce chapitre.

Ce chapitre clôture donc les travaux réalisés dans le cadre de cette thèse par l'exploration des dernières phases proposées dans l'approche générale d'HIDS présentée au Chapitre 3. Le chapitre suivant conclut ce manuscrit par un retour sur les travaux réalisés et les perspectives associées.

Conclusion

Bilan

Les dernières évolutions des systèmes avioniques cherchent à améliorer en continu l'expérience utilisateur tout en réduisant les coûts, que ce soit en terme de carburant, de développements, ou d'aspects environnementaux. Parmi ces évolutions, on peut noter l'introduction de nouvelles fonctionnalités, souvent liées à une plus grande connectivité des aéronefs. Dans un contexte d'évolution constante des menaces informatiques, et au regard de l'augmentation de la surface d'attaque résultant de l'augmentation de la connectivité sur les aéronefs, il devient primordial de considérer le problème de la sécurité des équipements avioniques. Des mesures doivent être envisagées et mises en place non seulement pour protéger les systèmes au niveau de leurs interfaces, mais aussi pour traiter des menaces plus subtiles provenant de l'intérieur des systèmes (tel qu'un acteur malveillant ou l'utilisation d'un équipement corrompu).

La difficulté majeure dans l'introduction de mesures de sécurité dans les systèmes avioniques réside dans les spécificités liées à ce type de système : embarqué, critique, temps-réel. Ce contexte particulier force les acteurs du domaine à adapter des concepts existants (par exemple, pour des systèmes d'information classiques), à des contraintes très spécifiques de l'embarqué critique temps-réel. En particulier, nous avons présenté les difficultés liées à l'introduction d'un HIDS sur un calculateur avionique sous la forme d'exigences relatives à l'efficacité de détection, la durée de vie, les performances en terme de ressources, l'impact sur l'exécution temps-réel de l'ensemble, l'impact sur la sûreté du vol, et la capacité à certifier un tel système.

Les travaux décrits dans ce manuscrit proposent et explorent une approche pour introduire un tel système de détection d'intrusion, au plus proche des applications avioniques, c'est-à-dire, directement sur un calculateur. Différents aspects de cette approche ont été abordés, notamment 1) l'introduction de la notion de *Domaine de Sécurité de l'Application* (SDA) pour représenter le périmètre de fonctionnement normal d'une application, 2) le développement d'un outil d'injection d'attaque pour application avionique, 3) une implémentation de l'approche sur une cible avionique représentative, et 4) des perspectives quant à l'analyse des alertes levées par l'HIDS afin de construire un message d'alerte porteur de suffisamment d'information pour être traité directement à bord et/ou être étudié au sol.

Plusieurs expérimentations ont été menées sur des applications avioniques réelles, permettant de mettre en évidence la pertinence de l'approche générale au regard des exigences fixées.

Dans un premier temps, l'approche prend en compte la durée de vie de l'aéronef en alliant un module de *Détection d'anomalie* avec un module de *Confirmation d'attaque*. La *Détection d'anomalie* est calibrée au plus proche du comportement normal de l'application, ce qui lui permet de rester efficace tout au long du cycle

de vie de l'application (qui sera très rarement mise à jour au cours de l'exploitation de l'aéronef). Ce calibrage est un point clé de l'approche, qui est réalisé lors de la phase d'intégration de l'application, avant son exploitation en opération. Pour ce calibrage, les tests et équipements existants sont réutilisés afin de limiter les coûts de développement de l'HIDS et de profiter de la maîtrise actuelle des applications avioniques, qui est nécessaire pour leur certification (notamment, la connaissance des différents modes de fonctionnement de l'application). Ensuite, la partie de *Confirmation d'attaque* permet d'une part de conserver une certaine flexibilité après la mise en exploitation de l'appareil (dans le cas où certains modes de fonctionnement de l'application n'auraient pas été pris en compte pour définir la partie de *Détection d'anomalie*), et d'autre part d'apporter un complément d'information suite à une alerte, qui pourra aider son traitement à bord ou un diagnostic plus poussé au sol.

Concernant l'implémentation embarquée, le prototype proposé utilise les principes de l'IMA afin de limiter l'impact de l'HIDS sur l'exécution temps-réel des applications et sur la sûreté du vol, tout en limitant le cadre de certification de l'HIDS. Plusieurs expérimentations ont également permis de montrer l'efficacité de détection de notre approche, et sa capacité à être intégrée sur une cible avionique sans engendrer un sur-coût trop important en terme d'utilisation des ressources.

Une première expérimentation a permis de valider le fait que l'outil d'injection d'attaque développé dans le cadre de ces travaux permettait effectivement de calibrer de façon efficace un HIDS avionique, sans connaissance préalable d'exemples d'attaques réelles. Dans ce cadre, une première version de l'HIDS a été calibrée avec des attaques émulées aléatoirement avant d'être évaluée face à des attaques menées de façon réaliste (réalisées manuellement, avec un impact réel sur l'application ciblée). Les résultats de cette évaluation ont montré que l'HIDS était capable de détecter l'ensemble des 9 attaques réalistes menées, alors que seules des attaques générées aléatoirement par l'outil (modifications réalisées de façon aléatoire pour émuler du code malveillant) ont été utilisées pour son calibrage.

Ensuite, pour évaluer plus précisément l'efficacité de détection de l'approche, plusieurs alternatives de SDA ont été proposées pour mettre en avant différentes exigences (par exemple, optimiser l'utilisation du CPU, de la mémoire, limiter les données observées, ou encore permettre une plus grande flexibilité pour l'implémentation embarquée), et ont été évaluées. Parmi elles, trois solutions ont été capable de détecter plus de 90% des attaques émulées tout en conservant un taux de 0% de faux positifs, sur deux applications distinctes utilisées comme cas d'étude. Deux de ces solutions ont ensuite été implémentées de façon réaliste sur un calculateur et ont permis de mettre en évidence leur capacité à détecter des attaques de façon exacte, en temps-réel et sans utiliser plus de 3% des ressources disponibles.

Enfin, une extension de la solution a été implémentée sur une cible embarquée afin de mettre en oeuvre la partie de *Confirmation d'attaque*, et ainsi caractériser les alertes levées par la partie de *Détection d'anomalie*. Les expérimentations menées à partir de cette implémentation ont donné des résultats encourageants quant à la possibilité de proposer un premier diagnostic en vol. En effet, nous avons été ca-

pables de donner un diagnostic exact pour plus de 90% des exemples de signatures d'alertes évaluées (plus de 700), en utilisant au préalable environ 500 exemples de signatures pour calibrer la partie de *Confirmation d'attaque*. De plus, la solution complète d'HIDS (*Détection d'anomalie* et *Confirmation d'attaque*) a présenté de très bons résultats en terme d'utilisation des ressources du calculateur, qui représente moins de 3.5% des ressources globales disponibles.

Ces travaux ont donc permis de valider l'intérêt d'une telle approche pour introduire un HIDS sur un calculateur avionique, et de donner des arguments en faveur de l'implémentation de compteurs de performance spécifiques à la sécurité dans les futurs calculateurs Thales. En particulier, un compteur sur les appels API a été implémenté sur les calculateurs actuellement en développement, en partie grâce aux résultats de ces travaux.

De façon plus générale, ces travaux ont fait l'objet de plusieurs publications [Damien 2018, Damien 2019a, Damien 2019b], dont deux se sont distinguées par des prix *Best Paper Award* (PRDC 2019) et *Best of Session (CSS-1) Award* (DASC 2019). Ces distinctions et les échanges avec d'autres industriels du domaine aéronautique (Airbus, Boeing, Rockwell Collins, Windriver) ou automobile (Mitsubishi) nous ont montré l'intérêt de la communauté pour ce sujet, et le besoin d'outils spécifiques de sécurité pour le domaine avionique, tel que l'outil d'injection d'attaque.

Limitations et perspectives

Il est important de noter certaines limitations aux travaux présentés dans ce manuscrit, et les perspectives envisagées pour pallier à ces limitations.

La première limitation concerne les caractéristiques des applications avioniques utilisées pour valider l'approche. En effet, ce sont des applications périodiques qui n'exposent qu'un seul mode de fonctionnement. Ce type d'application ne représente pas l'ensemble des applications avioniques possibles, qui peuvent par exemple exécuter des processus apériodiques (réaction à un évènement ou action pilote par exemple) ou être très complexes (nombreux modes de fonctionnement). Néanmoins, les applications utilisées ici restent représentatives de certaines applications très critiques telles que les commandes de vol. Il serait intéressant d'évaluer l'approche sur des applications plus complexes, et de la faire évoluer en proposant non pas un modèle de comportement normal pour l'ensemble de l'application, mais découper l'application en plusieurs modes de fonctionnement, et définir un modèle de comportement pour chacun de ces modes. Ce découpage pourrait être également effectué directement sur les processus de l'application, afin de prendre en compte plus simplement les processus apériodiques.

Ensuite, une hypothèse un peu forte a été prise en compte dans ces travaux : le fait que le fournisseur d'application est également capable de fournir les entrées d'activation permettant d'exécuter l'ensemble des modes de fonctionnement de l'application. Même si des tests fonctionnels sont effectués par le fournisseur

d'application, ils ne couvrent pas forcément tous les modes de fonctionnement de l'application. Certains modes sont effectivement couverts de façon théorique (par exemple, analyse statique de code), sans être explicitement testés avec une exécution sur cible. Dans ces travaux, la partie de *Confirmation d'attaque* pourrait être développée de façon beaucoup plus approfondie pour analyser la pertinence de l'approche face à des cas où un mode de fonctionnement particulier n'a pas été observé pendant la phase d'apprentissage de la *Détection d'anomalie*. Cette partie pourrait enrichir les entrées de la base de connaissances relatives aux fausses alertes, qui correspondraient dans ce cas à un mode de fonctionnement normal mais non pris en compte par la *Détection d'anomalie*.

L'outil d'injection d'attaque développé dans ces travaux présente également certaines limitations, même si la version proposée ici était suffisante dans le cadre de ces travaux. Dans un premier temps, l'utilisation de l'outil de debug GDB permet de conserver l'exécution temps-réel à l'intérieur du calculateur, mais n'agit pas sur l'environnement du calculateur. Dans le cas où le calculateur est stimulé par des actions extérieures (par exemples, des I/O ou via le réseau), cette implémentation n'est plus capable de conserver l'exécution temps-réel de l'ensemble. Par conséquent, il serait intéressant d'étudier d'autres technologies pour l'implémenter, afin d'appliquer l'outil sur un environnement complet (par exemple, plusieurs calculateurs en réseau) et pas seulement sur un calculateur isolé. Cela permettrait d'envisager plusieurs façons de stimuler l'application (entrées d'activation via les I/O ou le réseau par exemple), et d'envisager d'autres moyens de provoquer un comportement anormal de l'application surveillée (notamment, émuler une attaque aux interfaces provenant d'une autre application). Dans un second temps, la génération de la campagne d'injection d'attaques est effectuée de façon aléatoire mais on pourrait envisager d'améliorer cette génération afin de rendre les campagnes d'attaque plus pertinentes. Il serait par exemple intéressant d'étudier les techniques de *fuzzing* plus avancées, qui prennent en compte l'impact de chaque test pour choisir les prochains tests à effectuer (et ainsi, provoquer différents impacts en minimisant le nombre de cas de test réalisés). Dans notre contexte, l'observation des données envoyées sur le réseau ou des données affichées au pilote pourrait être une piste d'informations à prendre en compte pour orienter une campagne d'injection d'attaque à l'aide de ce type de technique.

Les travaux proposés ici se limitent également à l'observation des appels API ARINC 653 effectués par l'application surveillée. D'autres données ont été envisagées et évoquées afin de définir le modèle de comportement normal de l'application, mais n'ont pas été testés. Si l'observation seule des appels API a permis d'obtenir de bons résultats, deux expérimentations ont mis en évidence l'intérêt d'étudier également l'état de la mémoire afin de détecter et de caractériser plus précisément certains types d'attaque (qui consistent généralement à modifier des valeurs en mémoire). Il serait intéressant d'étudier les moyens d'observer l'utilisation de la mémoire et de corréler ces informations avec l'observation des appels API ARINC 653. En particulier, ce niveau d'observation pourrait être plus facilement envisa-

geable sur une architecture avec un processeur multi-coeurs, qui ne sont pas ou très peu utilisés aujourd'hui dans l'avionique mais pour lesquels des travaux sont en cours. Un coeur dédié à la surveillance des applications ouvrirait de nombreuses perspectives en terme d'observabilité des applications.

Pour finir, la réaction après une alerte n'a pas été abordée dans ces travaux, et représente un sujet très complexe à traiter. Néanmoins, cette réaction a été anticipée dans les choix qui ont été faits, en cherchant à utiliser des algorithmes simples et à donner du sens aux alertes levées. Il serait intéressant de confronter les résultats de détection obtenus dans ces travaux directement avec un avionneur ou un utilisateur (compagnie aérienne, pilote), pour étudier la faisabilité d'une réaction à bord à partir de ces alertes.

Bibliographie

[ANSSI 2012] ANSSI. *Maîtriser la SSI pour les systèmes industriels.* Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), juin 2012. (Cité en pages 26 et 30.)

[Arlat 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. . Fabre, J. . Laprie, E. Martins et D. Powell. *Fault injection for dependability validation : a methodology and some applications.* IEEE Transactions on Software Engineering, vol. 16, no. 2, pages 166–182, Feb 1990. (Cité en page 39.)

[Biesecker 2017] Calvin Biesecker. *Boeing 757 Testing Shows Airplanes Vulnerable to Hacking, DHS Says*, novembre 2017. (Cité en page 16.)

[Bochot 2009] Thomas Bochot, Pierre Virelizier, Helene Waeselynck et Virginie Wiels. *Model checking flight control systems : The Airbus experience.* Dans 2009 31st International Conference on Software Engineering - Companion Volume, pages 18–27, Vancouver, BC, Canada, 2009. IEEE. (Cité en page 27.)

[Boeing 2009] Boeing. *E-enabled capabilities of the 787 Dreamliner.* QTR_0109, Quarterly publication boeing.com/commercial/aeromagazine, 2009. (Cité en page 15.)

[Bonfante 2009] Guillaume Bonfante, Matthieu Kaczmarek et Jean-Yves Marion. *Architecture of a morphological malware detector.* Journal in Computer Virology, vol. 5, no. 3, pages 263–270, août 2009. (Cité en page 49.)

[Bonfante 2017] Guillaume Bonfante et Julien Oury Nogues. *Function Classification for the Retro-Engineering of Malwares.* Dans Frédéric Cuppens, Lingyu Wang, Nora Cuppens-Boulahia, Nadia Tawbi et Joaquin Garcia-Alfaro, editeurs, Foundations and Practice of Security, volume 10128, pages 241–255. Springer International Publishing, Cham, 2017. (Cité en page 49.)

[Buczak 2016] A. L. Buczak et E. Guven. *A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection.* IEEE Communications Surveys Tutorials, vol. 18, no. 2, pages 1153–1176, 2016. (Cité en page 36.)

[Butz 2007] Henning Butz. *The AIRBUS Approach to Open Integrated Mdoular Avionics (IMA) : Technology, Methods, Processes and Future Road Map.* page 11, 2007. (Cité en page 25.)

[Casals 2013] Silvia Gil Casals, Philippe Owezarski et Gilles Descargues. *Generic and Autonomous System for Airborne Networks Cyber-Threat Detection.* 32nd Digital Avionics Systems Conference (DASC), Syracuse, NY, page 14, 2013. (Cité en pages 30, 32 et 36.)

[Cerveira 2017] Frederico Cerveira, Raul Barbosa, Marta Mercier et Henrique Madeira. *On the Emulation of Vulnerabilities through Software Fault Injection.*

Dans 2017 13th European Dependable Computing Conference (EDCC), pages 73–78, Geneva, septembre 2017. IEEE. (Cité en pages 39 et 61.)

[Chandola 2009] Varun Chandola, Arindam Banerjee et Vipin Kumar. *Anomaly detection : A survey.* ACM Computing Surveys, vol. 41, no. 3, pages 1–58, juillet 2009. (Cité en page 36.)

[Chen 2011] T M Chen et S Abu-Nimeh. *Lessons from Stuxnet.* Computer, vol. 44, no. 4, pages 91–93, avril 2011. (Cité en page 15.)

[Condit 2007] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay et George C. Necula. *Dependent Types for Low-Level Programming.* Dans David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum et Rocco De Nicola, editeurs, Programming Languages and Systems, volume 4421, pages 520–535. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. (Cité en page 25.)

[Conmy 2003] Philippa Conmy, Mark Nicholson et John McDermid. *Safety Assurance Contracts for Integrated Modular Avionics.* page 10, 2003. (Cité en page 12.)

[Cooper 2017] Pete Cooper. Aviation Cybersecurity : Finding Lift, Minimizing Drag. novembre 2017. (Cité en pages 29 et 30.)

[Daley 2016] Brandon L Daley. *USBeSafe : Applying One Class SVM for Effective USB Event Anomaly Detection.* Rapport technique, Northeastern University, College ofComputer and Information Systems Boston United States, 2016. (Cité en page 36.)

[Damien 2018] A. Damien, M. Fumey, E. Alata, M. Kaâniche et V. Nicomette. *Anomaly based Intrusion Detection for an Avionic Embedded System.* Aerospace Systems and Technology Conference (ASTC), London, United Kingdom, Nov. 2018. (Cité en page 147.)

[Damien 2019a] A. Damien, N. Feyt, V. Nicomette, É. Alata et M. Kaâniche. *Attack Injection into Avionic Systems through Application Code Mutation.* Dans 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), septembre 2019. (Cité en page 147.)

[Damien 2019b] A. Damien, M. Marcourt, V. Nicomette, E. Alata et M. Kaâniche. *Implementation of a Host-Based Intrusion Detection System for Avionic Applications.* Dans 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), pages 178–17809, Dec 2019. (Cité en page 147.)

[Dessiatnikoff 2011] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaaniche et V. Nicomette. *A Clustering Approach for Web Vulnerabilities Detection.* Dans 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing, pages 194–203, Pasadena, CA, USA, décembre 2011. IEEE. (Cité en page 39.)

[Dessiatnikoff 2012] A. Dessiatnikoff, Y. Deswarte, É Alata et V. Nicomette. *Potential Attacks on Onboard Aerospace Systems*. IEEE Security Privacy, vol. 10, no. 4, pages 71–74, juillet 2012. (Cité en pages 39 et 63.)

[Dessiatnikoff 2013] A. Dessiatnikoff, V. Nicomette, É. Alata, Y. Deswarte, B. Leconte, A. Combes et C. Simache. *SEcuring Integrated Modular Avionics Computers*. Dans 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC), pages 4A3–1–4A3–11, Oct 2013. (Cité en pages 27 et 30.)

[Duraes 2006] Joao A. Duraes et Henrique S. Madeira. *Emulation of Software Faults : A Field Data Study and a Practical Approach*. IEEE Transactions on Software Engineering, vol. 32, no. 11, pages 849–867, novembre 2006. (Cité en pages 39 et 61.)

[Fonseca 2009] J. Fonseca, M. Vieira et H. Madeira. *Vulnerability #x00026 ; attack injection for web applications*. Dans 2009 IEEE/IFIP International Conference on Dependable Systems Networks, pages 93–102, juin 2009. (Cité en page 39.)

[Fumey 2018] Marc Fumey, Michael Templier et Christophe Mangion. *Method and electronic device for verifying a partitioning configuration, associated computer program*, octobre 2018. (Cité en page 25.)

[Gadelrab 2007] Mohammed S. Gadelrab, Anas Abou El Kalam et Yves Deswarte. *Defining categories to select representative attack test-cases*. Dans Proceedings of the 2007 ACM workshop on Quality of protection - QoP '07, page 40, Alexandria, Virginia, USA, 2007. ACM Press. (Cité en page 38.)

[Gandotra 2014] Ekta Gandotra, Divya Bansal et Sanjeev Sofat. *Malware Analysis and Classification : A Survey*. Journal of Information Security, vol. 05, no. 02, pages 56–64, 2014. (Cité en page 49.)

[Gatti 2016] Marc Gatti. *Évolution des Architectures des Systèmes Avioniques Embarqués*. page 219, juin 2016. (Cité en page 9.)

[Gil Casals 2014] Silvia Gil Casals. *Risk assessment and intrusion detection for airbone networks*. PhD Thesis, Toulouse, INSA, 2014. (Cité en pages 51, 53 et 79.)

[Glass-Vanderlan 2018] Tarrah R. Glass-Vanderlan, Michael D. Iannacone, Maria S. Vincent, Qian, Chen et Robert A. Bridges. *A Survey of Intrusion Detection Systems Leveraging Host Data*. arXiv :1805.06070 [cs], mai 2018. arXiv : 1805.06070. (Cité en pages 34 et 36.)

[Greenberg 2015] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. Wired, juillet 2015. (Cité en page 15.)

[Hay 2008] Andrew Hay, Daniel Cid et Rory Bray. Ossec host-based intrusion detection guide. Syngress Publishing, 2008. (Cité en page 31.)

[Huyck 2019] Patrick Huyck. *Safe and Secure Data Fusion – Use of MILS Multicore Architecture to Reduce Cyber Threats*. Dans 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), page 9, septembre 2019. (Cité en page 30.)

[IATA 2018] IATA. *IATA Releases 2018 Airline Safety Performance*, 2018. (Cité en page 16.)

[ISO/IEC 15408-1 2009] ISO/IEC 15408-1. *Information technology - Security techniques - Evaluation criteria for IT security.* 2009. (Cité en page 24.)

[Jacob 2008] Grégoire Jacob, Hervé Debar et Eric Filiol. *Behavioral detection of malware : from a survey towards an established taxonomy.* Journal in Computer Virology, vol. 4, no. 3, pages 251–266, août 2008. (Cité en page 49.)

[Jim 2002] Trevor Jim, Greg Morrisett, James Cheney, Dan Grossman, Michael Hicks et Yanling Wang. *Cyclone : A safe dialect of C.* USENIX Annual Technical Conference, General Track, pages 275–288, 2002. (Cité en page 25.)

[Kadar 2019] Marine Kadar, Sergey Tverdyshev et Gerhard Fohler. *System Calls Instrumentation for Intrusion Detection in Embedded Mixed-Criticality Systems.* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2019. (Cité en page 34.)

[Kim 2014] Gisung Kim, Seungmin Lee et Sehun Kim. *A novel hybrid intrusion detection method integrating anomaly detection with misuse detection.* Expert Systems with Applications, vol. 41, no. 4, pages 1690–1700, mars 2014. (Cité en page 32.)

[Klerx 2014] T. Klerx, M. Anderka, H. K. Büning et S. Priesterjahn. *Model-Based Anomaly Detection for Discrete Event Systems.* Dans 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, pages 665–672, novembre 2014. (Cité en page 36.)

[Kumar 1995] Sandeep Kumar et Eugene H. Spafford. *A software architecture to support misuse intrusion detection.* Computers & Security, vol. 14, no. 7, page 607, janvier 1995. (Cité en page 31.)

[Kwon 2017] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C. Suh, Ikkyun Kim et Kuinam J. Kim. *A survey of deep learning-based network anomaly detection.* Cluster Computing, septembre 2017. (Cité en page 35.)

[Laprie 1996] Jean-Claude Laprie, Jean Arlat, Jean-Paul Blanquart, Alain Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Hubert Guillermain, Mohamed Kaâniche, Karama Kanoun, Corinne Mazet, David Powell, Christophe Rabéjac et Pascale Thévenod. Guide de la sûreté de fonctionnement. Cépaduès, Toulouse (France), 1996. OCLC : 35123685. (Cité en page 6.)

[Les Echos 2015] Les Echos. *L'Agence européenne de sécurité aérienne alerte contre le risque de cyber-attaque.* Les Echos, octobre 2015. (Cité en page 1.)

[Liu 2017] Xiaoran Liu, Qin Lin, Sicco Verwer et Dmitri Jarnikov. *Anomaly Detection in a Digital Video Broadcasting System Using Timed Automata.* arXiv :1705.09650 [cs], mai 2017. arXiv : 1705.09650. (Cité en page 36.)

[Maglaras 2016] Leandros A. Maglaras, Jianmin Jiang et Tiago J. Cruz. *Combining ensemble methods and social network metrics for improving accuracy*

of OCSVM on intrusion detection in SCADA systems. Journal of Information Security and Applications, vol. 30, pages 15–26, octobre 2016. (Cité en page 36.)

[Marks 2019] Joseph Marks. *The Cybersecurity 202 : Hackers just found serious vulnerabilities in a U.S. military fighter jet.* Washington Post, août 2019. (Cité en page 16.)

[Maurer 2019] Nils Maurer, Thomas Graupl et Corinna Schmitt. *Evaluation of the LDACS Cybersecurity Implementation.* Dans 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), page 10, septembre 2019. (Cité en page 29.)

[MITRE 2018] MITRE. *MITRE ATT&CK™.* `https://attack.mitre.org/`, 2018. Accessed : 03-Apr-2019. (Cité en page 38.)

[Natella 2016] Roberto Natella, Domenico Cotroneo et Henrique S. Madeira. *Assessing Dependability with Software Fault Injection : A Survey.* ACM Computing Surveys, vol. 48, no. 3, pages 1–55, février 2016. (Cité en pages 39 et 58.)

[Om 2012] Hari Om et Aritra Kundu. *A hybrid system for reducing the false alarm rate of anomaly intrusion detection system.* Dans Recent Advances in Information Technology (RAIT), 2012 1st International Conference on, pages 131–136. IEEE, 2012. (Cité en page 32.)

[O'Neill 2016] K. O'Neill, G. R. Newell et S. K. Odiga. *Protecting flight critical systems against security threats in commercial air transportation.* Dans 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), pages 1–7, septembre 2016. (Cité en pages 29 et 30.)

[Or-Meir 2019] Ori Or-Meir, Nir Nissim, Yuval Elovici et Lior Rokach. *Dynamic Malware Analysis in the Modern Era—A State of the Art Survey.* ACM Computing Surveys, vol. 52, no. 5, pages 1–48, septembre 2019. (Cité en page 49.)

[Parkinson 2007] Paul Parkinson et Larry Kinnan. *Safety-Critical Software Development for Integrated Modular Avionics.* Rapport technique, Wind River, 2007. (Cité en pages 26 et 33.)

[PIPAME 2009] PIPAME. *Étude de la chaîne de valeur dans l'industrie aéronautique.* Technical Report, page 113, septembre 2009. (Cité en page 9.)

[Powell 2001] David Powell, Robert Stroud (editors, Sadie Creese (qinetiq, Yves Deswarte (laas cnrs, Klause Kursawe (ibm Zrl, Jean claude Laprie (laas cnrs, David Powell (laas cnrs et James Riordan (ibm Zrl. *Malicious- and accidental-fault tolerance for internet applications : Conceptual model and architecture.* 2001. (Cité en page 8.)

[Prisaznuk 2008] P. J. Prisaznuk. *ARINC 653 role in Integrated Modular Avionics (IMA).* Dans 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, pages 1.E.5–1–1.E.5–10, octobre 2008. (Cité en page 12.)

[Russell 2010] Stuart Jonathan Russell et Peter Norvig. Intelligence artificielle. Pearson Education, Paris, 2010. OCLC : 731521491. (Cité en page 35.)

[SAE 2019] ARINC Project SAE. *Internet Protocol Suite (IPS) for Aeronautical Safety Services*, 2019. (Cité en page 29.)

[Santamarta 2019] Ruben Santamarta. *Arm IDA and Cross Check : Reversing the 787's Core network.* IOActive white paper, août 2019. (Cité en page 16.)

[Scholkopf 2001] Bernhard Scholkopf, Robert Williamson, Alex Smola, John Shawe-Taylor et John Platt. *Support Vector Method for Novelty Detection.* page 7, 2001. (Cité en pages 51, 52, 79 et 101.)

[Snort 2019] Snort. *SNORT Users Manual 2.9.13.* The Snort Project, février 2019. (Cité en page 31.)

[Soulier 2015] Paul Soulier, Depeng Li et John R. Williams. *A survey of language-based approaches to Cyber-Physical and embedded system development.* Tsinghua Science and Technology, vol. 20, no. 2, pages 130–141, avril 2015. (Cité en page 24.)

[Studnia 2014] Ivan Studnia, Eric Alata, Vincent Nicomette, Mohamed Kaâniche et Youssef Laarouchi. *A language-based intrusion detection approach for automotive embedded networks.* Dans The 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015), Zhangjiajie, China, novembre 2014. (Cité en page 31.)

[Tabrizi 2015] Farid Molazem Tabrizi et Karthik Pattabiraman. *Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems.* Dans Dependable Computing Conference (EDCC), 2015 Eleventh European. IEEE, 2015. (Cité en pages 30 et 34.)

[Taft 2006] S. Taft, Robert Duff, Randall Brukardt, Erhard Ploedereder et Pascal Leroy. Ada 2005 reference manual. language and standard libraries - international standard iso/iec 8652/1995 (e) with technical corrigendum 1 and amendment 1, volume 4348. 01 2006. (Cité en page 25.)

[Teso 2013] Hugo Teso. *Aircraft Hacking - Practical Aero Series*, avril 2013. (Cité en page 15.)

[VanderLeest 2018] Steven H. VanderLeest. *Is formal proof of seL4 sufficient for avionics security ?* IEEE Aerospace and Electronic Systems Magazine, vol. 33, no. 2, pages 16–21, février 2018. (Cité en page 24.)

[Vapnik 2000] Vladimir Naumovich Vapnik. The nature of statistical learning theory. Statistics for engineering and information science. Springer, New York, 2nd ed édition, 2000. (Cité en page 51.)

[Vasilomanolakis 2016] Emmanouil Vasilomanolakis, Carlos Garcia Cordero, Nikolay Milanov et Max Muhlhauser. *Towards the creation of synthetic, yet realistic, intrusion detection datasets.* Dans NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, pages 1209–1214, Istanbul, Turkey, avril 2016. IEEE. (Cité en page 39.)

[Yoon 2013] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim et Lui Sha. *SecureCore : A multicore-based intrusion detection architecture for real-time embedded systems.* Dans Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th, pages 21–32. IEEE, 2013. (Cité en pages 31, 34 et 48.)

[Yoon 2015a] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu et Lui Sha. *Learning Execution Contexts from System Call Distributions for Intrusion Detection in Embedded Systems.* arXiv preprint arXiv :1501.05963, 2015. (Cité en page 34.)

[Yoon 2015b] Man-Ki Yoon, Lui Sha, Sibin Mohan et Jaesik Choi. *Memory heat map : anomaly detection in real-time embedded systems using memory behavior.* pages 1–6. ACM Press, 2015. (Cité en page 34.)

[Zennou 2018] Sarah Zennou, Saumya K. Debray, Thomas Dullien et Arun Lakhotia. *Malware Analysis : From Large-Scale Data Triage to Targeted Attack Recognition (Dagstuhl Seminar 17281).* page 10 pages, 2018. (Cité en page 49.)

Résumé :

Aujourd'hui, le transport aérien est l'un des modes de transport les plus sûrs, pour lequel les risques d'incidents depuis les débuts de l'aviation ne cessent de diminuer. Ces dernières décennies ont vu les systèmes avioniques évoluer (connectivité, partage de ressources, COTS) afin d'améliorer l'expérience passager et réduire les coûts. Si ces évolutions sont maîtrisées d'un point de vue *safety*, elles induisent néanmoins de nouveaux vecteurs d'attaque d'un point de vue *security*. Au regard des attaques récentes sur des systèmes embarqués ou critiques, il devient primordial d'anticiper ce type de menace pour l'avionique. Récemment, plusieurs études ont vu le jour concernant la sécurité des systèmes avioniques. La plupart se concentrent sur les interfaces de l'aéronef (moyens de communication ou de mises à jour logicielles) ou sur la phase de développement (analyses de risques, tests de vulnérabilités). Quelques travaux proposent des mesures de défense en profondeur (durcissement d'OS, détection d'intrusion), notamment pour se protéger d'attaquants internes. Dans cette thèse, nous prenons l'hypothèse qu'une application malveillante s'est introduite sur un calculateur avionique. Plus précisément, nous étudions donc la mise en place d'un système de détection d'intrusion au sein d'un calculateur avionique. Étant donné l'environnement considéré, nous avons formalisé six objectifs spécifiques relatifs à l'efficacité de détection, la durée de vie de l'aéronef, les performances, l'impact temps-réel, l'impact sur la sûreté, et la certification. Pour y répondre, nous proposons une approche complète permettant d'intégrer un système de détection d'intrusion sur un calculateur, en se basant sur le processus de développement IMA (*Integrated Modular Avionics*). Cette approche propose de modéliser le comportement normal d'une application avionique pendant la phase d'intégration, en s'appuyant sur les caractéristiques statiques et déterministes des applications avioniques, et sur les moyens déjà existants pour la *safety*. Ce modèle de comportement normal est ensuite embarqué à bord de l'aéronef et permet de détecter toute déviation de comportement pendant la phase d'opération. En complément, une fonction d'analyse d'anomalies embarquée offre un premier niveau de diagnostic à bord, et une certaine flexibilité une fois l'aéronef en opération. Cette approche a été implémentée sur deux cas d'étude afin de valider sa faisabilité et d'évaluer ses capacités de détection et sa consommation de ressources. Un outil d'injection d'attaque a été réalisé afin de pallier au manque de moyens existants pour tester notre approche. Plusieurs solutions de détection comportementale ont été proposées et évaluées, en se basant sur deux types de modèles : *OCSVM* et Automate temporisé. Deux implémentations sur calculateur embarqué ont permis d'observer de très bons résultats en termes d'efficacité de détection et d'utilisation des ressources. Enfin, l'implémentation de la fonction d'analyse d'anomalies et les expérimentations associées ont donné des résultats encourageants quant à la possibilité d'embarquer un tel système sur un aéronef.

Mots clés : Sécurité, Avionique, Détection d'intrusion, Apprentissage, IMA