Étude et conception de méthodes de protection face aux attaques par corruption de mémoire pour systèmes embarqués dans le contexte de l'Internet des Objets.

Yohan Boyer

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITE DE MONTPELLIER

En SyAM - Systèmes Automatiques et Micro-électroniques

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

Étude et conception de méthodes de protection face aux attaques par corruption de mémoire pour systèmes embarqués dans le contexte de l'Internet des Objets.

Présentée par Yohan Boyer

Le 17 décembre 2020

Sous la direction de Pascal Benoit

Devant le jury composé de

Pascal Benoit, Maître de Conférences, Université de Montpellier	Directeur de thèse
Jacques Bourhis, Ingénieur, IoTerop	Invité
Giorgio Di Natale, Directeur de recherche - CNRS, TIMA	Examinateur, Président
Guy Gogniat, Professeur, Université de Bretagne-Sud	Rapporteur
David Hely, Maître de Conférences, Grenoble-INP ESISAR	Rapporteur
Maria Mushtaq, Chercheur, Université de Montpellier	Examinatrice
Lionel Torres, Professeur, Université de Montpellier	Examinateur

UNIVERSITÉ DE MONTPELLIER

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

I

Introduction

Sommaire

1.1 Contexte - Internet des Objets

1.1.1 Définition de l'Internet des Objets (IdO)

Le terme Internet des Objets (IdO) est apparu pour la première fois en 1999 dans un discours de Kevin ASHTON, un ingénieur britannique de chez Procter & Gamble. Il servait à désigner un système où les objets physiques sont connectés à Internet et capables de créer et transmettre des données afin de créer de la valeur pour ses utilisateurs. C'est en 2012 que la définition de l'IdO sera donnée par l'Union Internationale des Télécommunications (UIT) : *"une infrastructure mondiale pour la société de l'information, qui permet de disposer de services évolués en interconnectant des objets (physiques ou virtuels) grâce aux technologies de l'information et de la communication interopérables existantes ou en évolution"*. En réalité, la définition de l'IdO n'est pas figée. Elle recoupe des dimensions d'ordres conceptuels et techniques. D'un point de vue conceptuel, l'IdO caractérise des objets physiques connectés ayant leur propre identité numérique et capables de communiquer les uns avec les autres. Ce réseau crée en quelque sorte une passerelle entre le monde physique et le monde virtuel. D'un point de vue technique, l'IdO consiste en l'identification numérique, à travers différents protocoles, d'un objet physique grâce à un système de communication, généralement sans fil, pour transférer des données et créer des services à valeurs ajoutées. Dans une étude récente, Bruno Dorsemaine *et al.* ont développé la définition et la taxonomie de l'Internet des Objets [2]. Tous ces points créent l'écosystème de l'IdO et définissent les bases de son architecture.

1.1.2 IdO : Architecture, réseau et caractéristiques

L'écosystème de l'IdO (figure 1.1) peut être découpé en deux mondes bien distincts. D'une part, on trouve le monde du Cloud avec l'Internet, les centres de données et les utilisateurs. Ce monde communique généralement à travers des protocoles IP. D'autre part, on trouve le monde embarqué, qui est composé d'objets connectés, et qui communique généralement à l'aide de radios. Ces deux mondes sont connectés à travers des passerelles qui permettent de passer d'un monde IP à un monde radio et vice versa. Les objets connectés sont des équipements actifs ou passifs pouvant générer des données ou directement agir sur l'environnement. Ils sont composés de capteurs, d'actionneurs et d'un ou plusieurs éléments permettant la transmission de ces données. De nombreux réseaux sans fil sont utilisés dans l'IdO, on y trouve les réseaux WIFI, Bluetooth, des réseaux opérés M2M comme la 3G et la 4G, ou encore des réseaux LPWAN comme Sigfox et LoraWAN. Ces réseaux apportent chacun leurs avantages et leurs inconvénients comme la portée de la communication, la consommation, la bande passante, le coût d'abonnement, ou encore la couverture. Il en résulte que, quel que soit le type de réseau utilisé, l'objet connecté génère de la valeur, que ce soit à travers la collecte de données ou l'action sur l'environnement, et c'est à la passerelle de s'occuper des transmissions entre ces objets et l'utilisateur final dans le monde IP.

Toutes ces possibilités ont amené à l'explosion de l'IdO. Tous les secteurs veulent

aujourd'hui déployer des objets connectés et ont des besoins et des applications spécifiques. En logistique, il peut s'agir de capteurs qui servent à la traçabilité des biens pour la gestion des stocks et les acheminements. Dans le domaine de l'environnement, il est plutôt question de suivi de la qualité de l'air, la température, du niveau sonore, ou encore de l'état d'un bâtiment. En domotique, l'IdO recouvre tous les appareils électroménagers communicants, les capteurs (thermostat, détecteur de fumée, de présence), les compteurs intelligents (eau, électricité, gaz), ou encore les systèmes de sécurité. On retrouve aussi l'IdO dans la surveillance des équipements industriels pour la maintenance prédictive, la gestion automatisée d'alertes, l'optimisation de processus, ou encore l'amélioration des rendements et des approvisionnements. Le phénomène IdO est également très visible dans le domaine de la santé et du bien-être avec le développement des montres connectées, des bracelets connectés et d'autres capteurs surveillant des constantes vitales. Selon le cabinet IoT Analytics, le déploiement d'objets connectés était de 17,8 milliards en 2018. La croissance devrait se poursuivre de plus belle pour atteindre les 34,2 milliards en 2025. La multitude d'applications, d'objets et de services crée donc de nouveaux besoins et problèmes. En effet, avec une telle masse d'objets, des questions de gestion d'objets et de sécurité se posent.

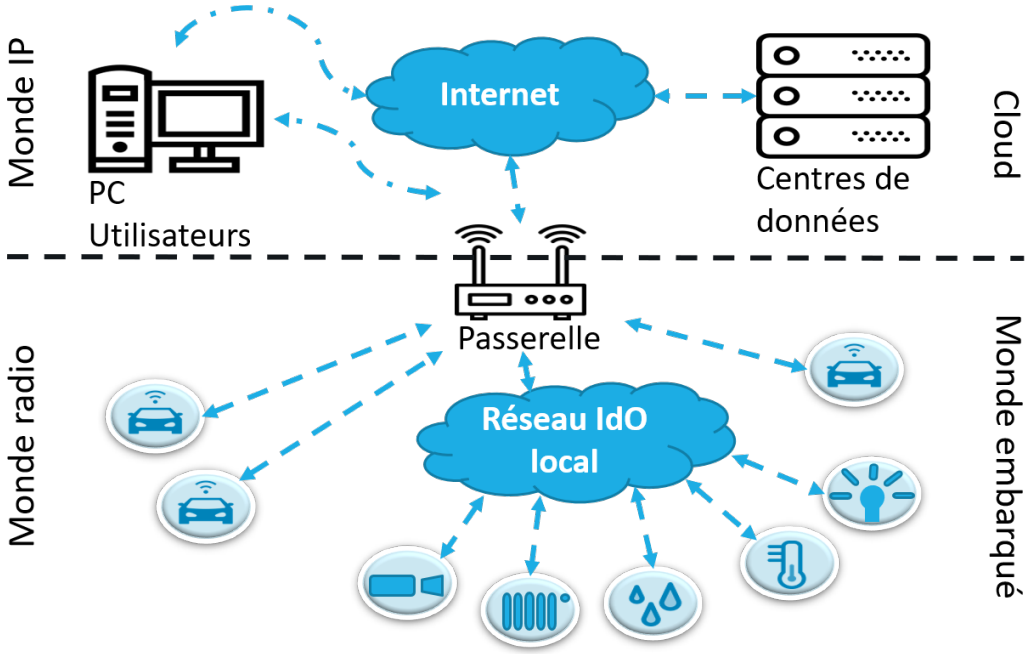

FIGURE 1.1 – *Architecture de l'Internet des Objets.*

1.2 Enjeux

1.2.1 Gestion des objets et sécurité

Avec le développement massif de l'Internet des Objets, de gros problèmes d'interopérabilité se posent [3]. Il y a aujourd'hui énormément de plateformes IdO différentes développées par de grands acteurs du numérique. Chacun développe son propre système, ses propres protocoles, et tout est souvent propriétaire. Cela pose des problèmes

d'intégration et d'interopérabilité lorsqu'un utilisateur veut interagir avec des objets d'un fabricant différent par exemple. De plus, de nombreuses études ont montré qu'il y a un vrai problème de sécurité dans l'IdO. Alaba *et al.* ont réalisé une enquête sur la sécurité dans l'IdO [4] qui définit les grands axes. Les auteurs offrent une vision de la taxonomie de la sécurité sur l'Internet des Objets, découpée en quatre catégories: Application, Architecture, Communication, et Données. Chacune de ces catégories possède un sous ensemble de vulnérabilités spécifique. Plus récent, en 2019, Neshenko *et al.* font un état de l'art sur la sécurité dans l'IdO et expriment les manques actuels [5]. Il est fait état des problèmes de déploiement de sécurité comme le coût, la consommation, ou encore le manque de standards. L'enquête révèle que les vulnérabilités peuvent se placer sur trois couches: une couche matérielle axée sur le dispositif, une couche réseau et une couche applicative axée sur le logiciel.

1.2.2 Des vulnérabilités à différentes couches

La *IoT Security Foundation* [6] affirme que moins de 10% des produits IoT sur le marché sont conçus avec des mécanismes de sécurité suffisants, et ce problème est très largement documenté dans la presse et les publications scientifiques [7, 8, 9]. Que ce soit des éléments du monde Cloud avec les PC, les centres de données, ou les éléments du monde embarqué avec tous les objets connectés, ou même les passerelles qui font le lien entre ces deux mondes, ils sont tous composés de trois grandes couches: une couche matérielle, une couche réseau et une couche logicielle. La couche logicielle peut aussi être décomposée en plusieurs parties: système d'exploitation, drivers, applications. Tous ces points laissent place à des vulnérabilités, plus ou moins critiques. C'est encore plus juste pour les objets connectés qui subissent des contraintes de consommation et d'autres limitations comme le coût, et de ce fait sont moins sécurisés qu'un centre de données par exemple. Nous allons voir que chaque couche contient ses propres vulnérabilités.

1.2.2.a Couche matérielle

La couche matérielle comprend tous les éléments physiques du système, que ce soit le processeur, les mémoires ou même les radios. De par sa nature, les vulnérabilités que l'on trouve sur la couche matérielle sont difficilement corrigeables sur les dispositifs déjà fabriqués. Un des grands problèmes de la couche matérielle des objets connectés est qu'un attaquant peut avoir un accès physique à ce dernier. À partir de là, il est capable de lancer des attaques par canaux cachés [10, 11, 12] pour retrouver des informations sensibles. Il peut aussi opérer un déni de service ou directement récupérer des informations sensibles dans la mémoire comme des mots de passe ou des clés de sécurité [13, 14]. Il peut ensuite utiliser ces informations pour répliquer un objet sur le réseau [15]. Les axes d'attaques sont très vastes de par l'accès physique et au final très compliqués à protéger.

1.2.2.b Couche réseau

La couche réseau englobe toutes les communications entre les systèmes. Dans le cas des objets connectés, cette communication est faite par radio. Plusieurs types de radios existent, et il y a énormément d'attaques sur chacune d'entre elles. Les travaux de Abdul-Ghani *et al.* ont fait un état de l'art sur ces attaques [16]. Sur la couche réseau on retrouve aussi des attaques *man-in-the-middle* (MITM) qui ont pour but d'intercepter des données ou d'usurper une identité [17]. Des problèmes de contrôle d'accès et d'authentification existent aussi au niveau réseau [18]. Dans la même idée, de nombreux problèmes et de limitations sont présents avec les services de gestion de clés de sécurité (KMS) [19]. Encore une fois, la variété de radios qui existe laisse une surface d'attaque d'autant plus grande. De plus, des problèmes de gestion de clés de sécurité se posent, dans un écosystème où comme nous avons vu, il est difficile de garder des secrets.

1.2.2.c Couche logicielle

La couche logicielle contient le système d'exploitation, les drivers et les applications. Cette couche exploite les deux précédentes pour mettre en place des services et générer de la valeur à travers l'objet connecté. Elle n'est cependant pas exempte de failles. Par exemple, un des botnets les plus connus qui a fait couler beaucoup d'encre suite à des attaques distribuées de déni de service s'appelle Mirai [20]. Un botnet est un réseau de machines informatiques considérées comme des robots, connectées à Internet, ayant pour but d'exécuter certaines tâches. Les appareils infectés par Mirai recherchent en permanence à atteindre de nouveaux objets connectés, et tentent de s'y connecter grâce à une table d'identifiants et de mots de passe par défaut connus afin d'y installer le logiciel malveillant. Une fois Mirai installé, les objets continuent de fonctionner normalement, malgré une lenteur occasionnelle due à une augmentation de l'utilisation de la bande passante. C'est ainsi que plusieurs attaques par déni de service ont été exécutées en septembre 2016, dont une atteignant plus de 600 Gbps de trafic réseau. La corruption d'un objet par fuite de mots de passe ou d'identifiants n'est pas la seule méthode. On retrouve aussi des corruptions mémoires logicielles de type dépassement de tampon [21]. Cette famille d'attaques fait partie des plus dévastatrices dans le monde logiciel et est continuellement exploitée depuis les années 1990. D'autres travaux ont montré différentes techniques pour faire fuiter des informations confidentielles relatives aux systèmes d'exploitation et aux données transférées [22, 23, 24, 25].

En somme, l'IdO possède de nombreux vecteurs d'attaques à différents niveaux d'implémentation. Cette variété de vulnérabilités, de problèmes et de limitations demande un effort de la communauté pour travailler ensemble et ainsi adresser un maximum de problèmes et trouver un maximum de solutions. Dans la suite de la thèse, nous avons fait le choix de travailler sur la couche logicielle, nous justifierons ce choix dans le chapitre 2, consacré à la définition d'attaques par corruption de mémoire.

1.3 Contexte de la thèse

1.3.1 Acteurs

Les travaux présentés dans ce manuscrit de thèse ont été effectués dans le cadre du dispositif Convention Industrielle pour la Formation par la REcherche (CIFRE). Le principe de ce dispositif est la collaboration entre une entreprise et un laboratoire en vue d'apporter des solutions innovantes dans le monde de la recherche.

1.3.1.a IoTerop

Fondée en 2016, IoTerop est une startup qui propose des solutions de gestion d'objets, d'interopérabilité et de sécurité pour l'Internet des Objets. La startup est spécialisée dans le LightWeigth Machine to Machine (LWM2M) [26], un protocole standardisé spécialement conçu pour l'IdO. Son offre repose aujourd'hui sur deux principaux axes:

1. Une offre qui cible les objets connectés avec une proposition d'implémentation du LWM2M légère, rapidement portable et sécurisée.

2. Une offre côté Cloud qui propose un serveur LWM2M avec tout un panel de services visant à gérer les flottes d'objets déployées.

Le LWM2M est un protocole très intéressant, car il adresse plusieurs problèmes rencontrés dans l'IdO en même temps. Il fixe les problèmes d'interopérabilité entre les objets et les fabricants. Il permet aussi une gestion des flottes déployées très facile grâce à des objets standards prédéfinis et des API standardisées. Il va aussi adresser des problèmes de sécurité tels que l'authentification et la cryptographie. Cependant, il reste des points noirs en termes de sécurité que le protocole ne peut/va pas gérer de lui-même. C'est dans ce cadre que la thèse a été mise en place.

1.3.1.b Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)

L'équipe ADAptive Computing (ADAC) est une équipe du département Microélectronique au LIRMM. Elle réunit des chercheurs et enseignants-chercheurs aux compétences complémentaires réunis autour de la thématique fédératrice de la conception de systèmes adaptatifs, composants capables de s'autogérer pour optimiser leurs performances au sens de propriétés diverses (puissance de calcul, consommation, fiabilité, sécurité). Les domaines d'application adressés sont larges, et concernent plus spécifiquement les systèmes embarqués, l'internet des objets (IoT) et les grilles de calcul.

1.3.2 Objectifs

La première étape de la thèse est de mettre en place un modèle de menace afin de définir nos axes de travail sur la sécurité des systèmes embarqués. L'objectif est de sécuriser de bout en bout les données et les équipements. L'analyse du modèle de menace, exhaustif au possible, doit permettre de définir des priorités dans les failles potentielles, afin de cibler un sous-ensemble réaliste dans le cadre de la thèse, fixant les niveaux de sécurité à atteindre. Une fois ce sous-ensemble identifié, un état de l'art doit être fait, concernant aussi bien les attaques que les protections relatives à ce sous-ensemble. Sur la base de cet état de l'art, il faudra comprendre comment fonctionnent ces attaques, comprendre comment fonctionnent les protections, mais aussi déterminer leurs limitations. Grâce à cette étude, nous pourrons être force de proposition pour contribuer à l'amélioration de la sécurité dans l'IdO et intégrer ces nouvelles solutions.

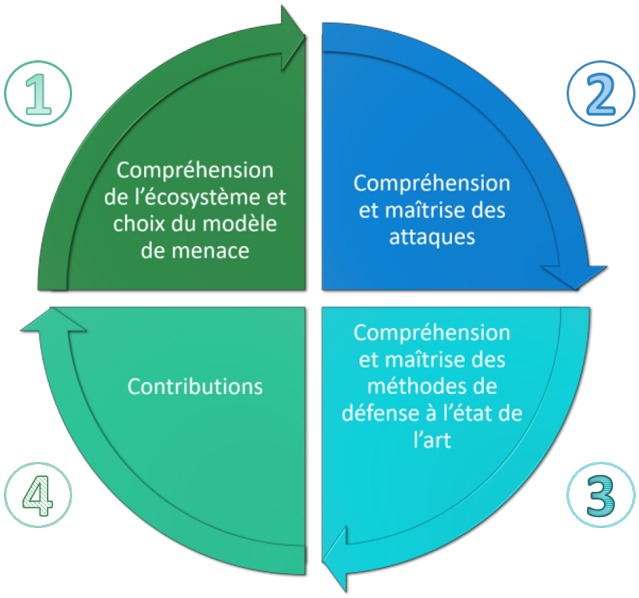

FIGURE 1.2 – *Axes de la thèse*

La figure 1.2 illustre le cheminement de la thèse. L'analyse du modèle de menace nous a mené aux attaques par corruption de mémoire. IoTerop étant expert dans le protocole du LWM2M, nous avons écarté certaines vulnérabilités bien connues dans l'IdO. En effet, le LWM2M traite déjà en partie les cas de mauvaises authentifications ou encore de mauvaises gestions cryptographiques. Un cas non traité, mais non moins critique, sont les attaques par corruption de mémoire. En effet, une simple vulnérabilité dans le code d'un logiciel peut permettre à un attaquant de prendre un contrôle total de ce dernier. Cette famille d'attaques est un fléau depuis les années 1990, et un rapport récent de Microsoft indique que le pourcentage de failles par corruption de mémoire entre 2006 et 2018 n'a pas changé et reste autour des 70% [27], figure 1.3. Nous proposons donc d'évaluer les vulnérabilités, les attaques et les protections liées aux corruptions de mémoire.

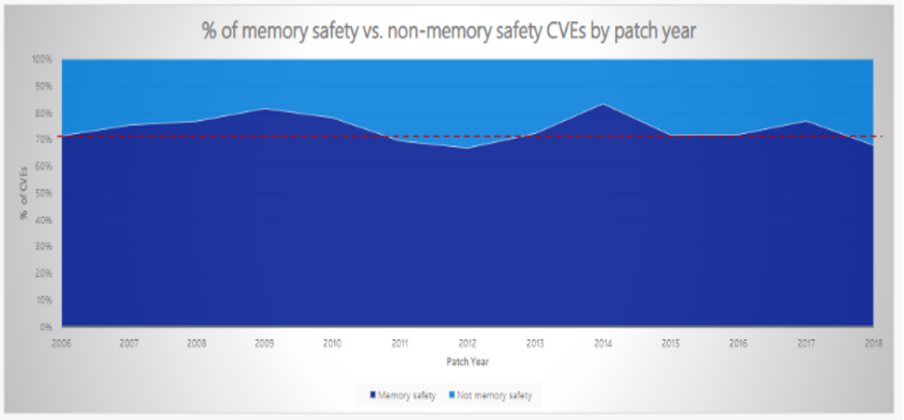

FIGURE 1.3 – *Rapport de Microsoft sur le taux d'attaques par corruption de mémoire.*

1.3.3 Structure du manuscrit

Ce manuscrit est organisé en six chapitres :

- Le premier, celui-ci, fait office d'introduction du sujet dans son contexte.
- Le deuxième chapitre se concentre sur une famille d'attaques bien connue depuis les années 2000 : les attaques par corruption de mémoire. Le but de ce chapitre est d'apporter tout le bagage technique nécessaire à la compréhension de ces attaques et à leur mise en place.
- Le troisième chapitre propose un état de l'art sur les protections contre les attaques par corruption de mémoire. Une classification des différentes méthodes est proposée. Ce chapitre permet de bien comprendre comment fonctionnent les protections actuelles, leurs limitations et pourquoi les attaques de cette famille sont toujours un fléau.
- Le quatrième chapitre propose une méthode de détection d'attaques pendant l'exécution basée sur le comportement normal d'un programme, l'utilisation de compteurs de performances matériels et la mise en place de machine learning. Une étude poussée sur le choix des évènements et des modèles de machine learning est proposée, puis de nombreuses expérimentations et résultats sont mis en avant.
- Le cinquième chapitre propose d'étudier le déroulement complet du rétablissement d'un système après la détection d'une attaque. Dans un premier temps, nous proposons une méthodologie pour améliorer la précision des modèles de machine learning à travers un diagnostic du système lors d'une alerte. L'idée est de réduire le taux de faux positifs. Dans un second temps, nous proposons d'évaluer la sécurité apportée par les méthodes de détection d'attaques combinées aux méthodes de contremesure. Dans une troisième partie, nous proposons un mécanisme de rétablissement robuste reposant sur des mécanismes de réinitialisation et de redémarrage.
- Enfin, le dernier chapitre conclut sur les travaux effectués durant ces trois années de thèse et propose des ouvertures et perspectives sur les différents sujets traités.

II

Les attaques par corruption de mémoire

Sommaire

2.1 Motivations

L'exploitation de vulnérabilités de corruption de mémoire est une menace depuis plus de trois décennies, et aucune contremesure ne semble pouvoir y mettre fin. Comme nous venons de voir, l'univers informatique est varié : d'un ordinateur à usage personnel à des serveurs d'une grande complexité demandant de répondre à des milliers de requêtes en même temps, sans oublier les téléphones connectés et plus généralement l'Internet des Objets (IdO) [28]. Cette croissance rapide des systèmes informatiques et de l'IdO avec une forte interconnexion par l'Internet donne aux attaquants un champ de plus en plus large pour exploiter ces dispositifs. Les attaques par corruption de mémoire peuvent être effectuées à distance et ne nécessitent pas forcément d'accès physiques aux cibles. Ces bogues proviennent de logiciels écrits dans des langages permissifs sur la gestion mémoire tels que le C, le C++, l'Objective-C, ou encore l'interpréteur Java. Ces langages permettent au programmeur de traiter des éléments de bas niveau tels que les pointeurs, la taille des tampons et les différentes sections dans un binaire. Cela peut conduire à différentes vulnérabilités à l'intérieur du binaire permettant à un attaquant de les exploiter. L'aspect intéressant de cette famille de vulnérabilité c'est qu'elle va cibler aussi bien un petit objet connecté qu'un serveur à plusieurs milliers d'euros. Même si différentes attaques peuvent exploiter différents bogues, l'objectif principal reste le même : rediriger le flot de contrôle du programme pour exécuter un nouveau morceau de code, *i.e.*, un code malveillant écrit par l'attaquant. Il peut conduire à l'exfiltration de données, à l'escalade de privilèges ou à l'exécution de code arbitraire (ACE) qui peut être transformé en exécution de code à distance (RCE). De nos jours, la majorité des bogues de sécurité relève encore des problèmes de sécurité dus à la gestion de la mémoire. Par exemple, en 2018, Microsoft a indiqué que 70% de tous les bogues de sécurité sont des problèmes de gestion de la mémoire [27]. En 2017, les ShadowBrokers, un groupe de hackeurs, ont volé des programmes d'exploitation fournissant des RCE à la NSA [29]. Parmi eux, Eternalchampion et Eternalsystem exploitent une RCE, EternalBlue exploite une RCE via SMB (Server Message Block), ExplodingCan exploite une RCE via IIS (Internet Information Service) 6.0. Cette fuite a conduit au premier logiciel de rançon largement répandu : WannaCry [30]. Les dommages causés par ce logiciel ont coûté plus d'un milliard de dollars [31]. C'est donc avec ces données en tête que nous choisissons de travailler sur les corruptions de mémoire et que nous passons à l'étape 2 sur la figure des axes de la thèse 2.1: "Compréhension et maîtrise des attaques". En effet, pour pouvoir proposer de nouvelles méthodes de protection ou améliorer les actuelles, il faut d'abord bien comprendre comment fonctionnent ces attaques. Pour répondre à ce besoin, une étude du fonctionnement d'un programme sur un processeur et des problèmes de management de mémoire est proposée.

2.2 Un lien logiciel/matériel

Pour bien comprendre comment les différentes parties d'un système s'interconnectent lors de l'exécution d'un programme, il est important de comprendre comment fonctionnent ces différentes parties une à une.

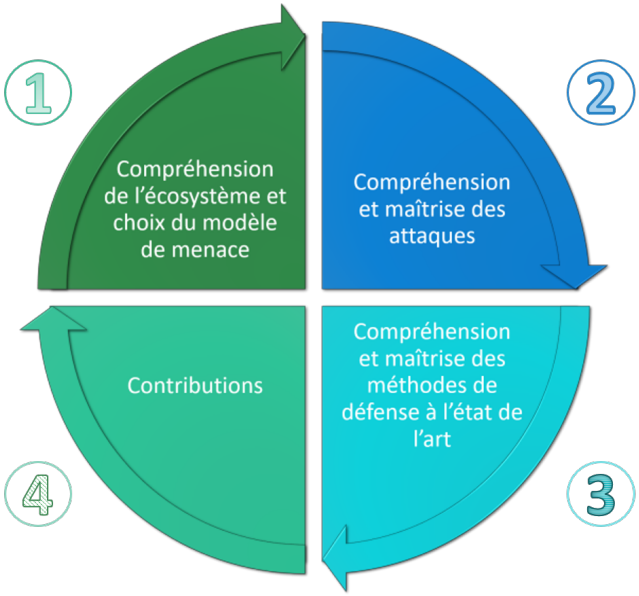

FIGURE 2.1 – *Axes de la thèse*

2.2.1 Langage de programmation

Un langage de programmation est une notation syntaxique pour écrire des algorithmes et produire des programmes machines de manière compréhensible. Ces langages vont permettre de créer ce que l'on appelle le code source d'un logiciel. Il existe énormément de langages de programmation différents et chacun a ses particularités. Certains sont considérés comme étant des langages de haut niveau et ciblent plutôt des machines performantes, car ils reposent sur des interpréteurs, *i.e.*, Java, Python ou JavaScript, alors que d'autres sont définis comme des langages de bas niveau, *i.e.*, C ou Assembleur, et ciblent plutôt des systèmes qui se veulent légers comme dans l'IoT. Ces langages de bas niveau sont ce que l'on appelle des langages compilés. C'est-à-dire qu'une fois que le code source est écrit, un compilateur est utilisé pour le transformer en code machine. Le compilateur dépend de l'architecture ciblée. Bien que ces deux catégories, *i.e.*, haut niveau et bas niveau, permettent de générer du code machine, les langages de bas niveau vont généralement proposer à l'utilisateur une gestion plus fine de la mémoire. Cette gestion lui permet de mieux optimiser son programme pour qu'il soit plus léger, plus rapide, mais d'un autre côté, cela donne aussi des possibilités d'erreurs dans le management de la mémoire. Ces langages sont dits permissifs. On y retrouve le C et l'Assembleur, mais aussi des langages tels que le C++ ou l'Objective-C qui sont définis comme des langages de haut niveau, mais qui permettent une gestion fine de la mémoire pour améliorer leurs performances. Les interpréteurs de langage haut niveau comme Java ou JavaScript peuvent aussi être sujets à des problèmes de management de mémoire. Cette fois, ce n'est pas le code source généré par ces langages qui peut contenir une vulnérabilité mais directement l'interpréteur car il est écrit à l'aide de langages permissifs. Dans ces travaux, nous prenons comme exemple l'utilisation du langage C pour plusieurs raisons. D'une part, c'est le langage le plus classique dans la programmation de systèmes embarqués. D'autre part, c'est le langage le plus compréhensible, le plus interopérable et le plus permissif parmi ces exemples. Pour bien comprendre les types de problèmes qui peuvent arriver à cause d'une ges-

tion fine de la mémoire, nous allons expliquer comment cette dernière fonctionne.

2.2.2 Management de mémoire

Dans un système informatique, il existe deux types principaux de mémoires : la mémoire vive (RAM) et la mémoire de masse (stockage). La mémoire de masse sert à stocker tous les fichiers que l'utilisateur utilise sur un ordinateur par exemple. Par opposition, la mémoire vive sert à stocker temporairement les fichiers que l'ordinateur exécute. De ce fait, c'est aussi la RAM qui contient tout le contexte d'exécution d'un programme. Ce contexte change en fonction de la couche logicielle ciblée, mais aussi du système d'exploitation (OS) utilisé. Dans ces travaux, nous prenons comme exemple la couche applicative, car plus généraliste et plus simple à comprendre, et nous sélectionnons l'OS Linux car très utilisé dans le monde embarqué. Les concepts qui sont expliqués restent similaires quel que soit l'OS ou la couche logicielle.

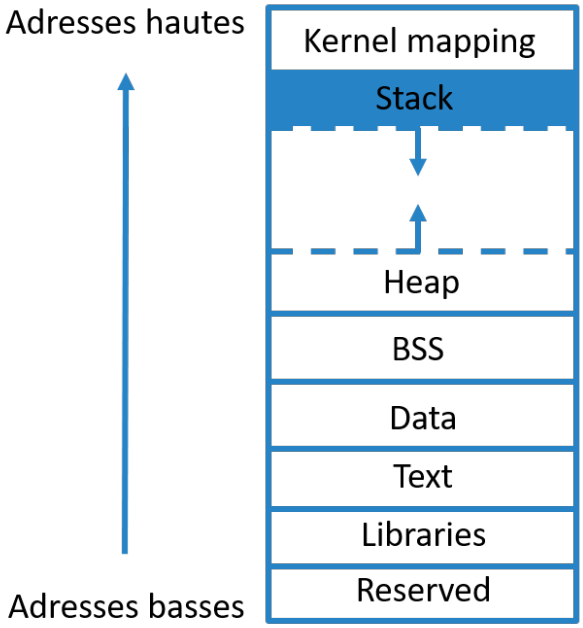

FIGURE 2.2 – *Découpage en section d'un programme au format ELF.*

Un format de fichier pour les codes compilés bien connu sur Linux est le format ELF [32]. Chaque fichier ELF est constitué d'un entête fixe, de segments et de sections. Les segments contiennent les informations nécessaires à l'exécution du programme, alors que les sections contiennent les informations pour la résolution des liens entre fonctions et les données. La figure 2.2 illustre les principales sections qui nous intéressent. Dans l'ordre, on y trouve une section pour le noyau (**Kernel mapping**), deux sections utiles au fonctionnement du programme pour stocker ses données d'exécution (la **Stack** et la **Heap**). Les sections **BSS** et **Data** contiennent respectivement les données non initialisées et les données initialisées du binaire. Ensuite vient la section **Text**, qui représente toutes les instructions du programme, puis la section **Libraries** qui comme son nom l'indique contient une cartographie des librairies utilisées. Lors de l'exécution, l'OS utilise toutes ces sections pour exécuter l'application. Le langage C permet donc de gérer finement le lien entre ces sections et les sections elles-mêmes.

C'est ce contrôle fin qui introduit potentiellement des vulnérabilités à l'intérieur du programme, et permet à un attaquant d'altérer le flot de contrôle de l'application.

2.2.3 Le flot de contrôle d'un programme

Le flot de contrôle d'un programme est l'ordre d'exécution des instructions. Ce flot est construit lors de la compilation du code source pour former le binaire du programme. Un binaire est constitué de plusieurs instructions. Certaines des instructions vont avoir un rôle plutôt arithmétique alors que d'autres vont directement jouer sur le flot d'exécution du programme. C'est le cas des instructions *ret*, *call* et *jmp*. Cette liste est non exhaustive, car il existe des multitudes de variantes qui dépendent aussi de l'architecture du processeur. Ces instructions permettent de construire ce qu'on appelle des Blocs de Base (BB). Un BB est composé d'un point d'entrée et d'un point de sortie. Le point de sortie fait forcément partie de la catégorie des instructions pouvant changer le flot d'un binaire. Entre ces deux points, on retrouve seulement des instructions qui ne peuvent pas changer ce flot. Le graphe du flot de contrôle (CFG) d'un programme est une succession de blocs de base. Un exemple est donné dans la figure 2.3. La succession et l'ordre d'enchaînement des BB sont définis à la compilation. Un CFG est donc fixe dans le temps et non altérable.

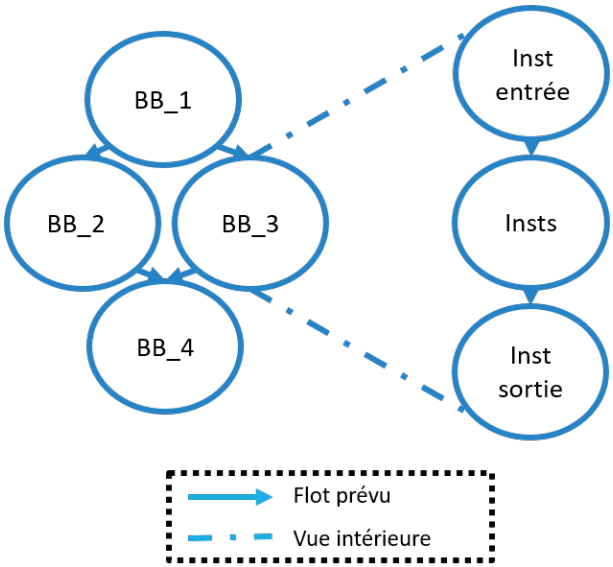

FIGURE 2.3 – *Graphe de flot de contrôle (CFG).*

Cependant, les programmes possèdent généralement des liens avec l'extérieur, par exemple des entrées utilisateurs, des périphériques ou même des moyens de communication. Ces liens laissent une porte ouverte à un utilisateur malintentionné pour venir essayer d'attaquer le programme et d'altérer le flot d'exécution du binaire. Ces attaques sont généralement nommées: "attaque par corruption de mémoire". L'attaquant cherche une erreur dans la gestion de la mémoire due à l'utilisation du langage C par exemple. Il cherche ensuite une entrée pour venir altérer ce management mémoire en utilisant les différents concepts vus précédemment, en outre la gestion des sections. Il est ensuite en capacité de venir rediriger le flot d'exécution du programme pour exécuter son propre code malicieux. Le premier ver bien connu exploitant une vulnérabilité

issue d'une corruption mémoire était le ver de Morris [33]. Cette attaque exploitait un dépassement de tampon que nous détaillons dans la prochaine section.

2.2.4 Dépassement de tampon

Le ver Morris était un ver informatique distribué via l'Internet, écrit par Robert Tappan Morris, lancé le 2 novembre 1988. Il exploitait deux vulnérabilités connues dans *sendmail* et *fingerd*. Une de ces deux failles était un dépassement de tampon de l'utilitaire *finger*. Le ver utilisait cette faille parmi d'autres pour se propager et avait pour but de sonder l'Internet de l'époque. Plus tard, en 2001, un nouveau ver fit son apparition: Code Red [34]. Il exploitait un dépassement de tampon présent dans le serveur Web IIS de Microsoft. Son but originel était de prendre le contrôle de ces serveurs pour lancer à partir d'eux une attaque par déni de service (DoS). Ces deux vers ont montré le fort impact que peut avoir un dépassement de tampon. Pour bien comprendre cet impact, il est intéressant de voir comment fonctionne un dépassement de tampon.

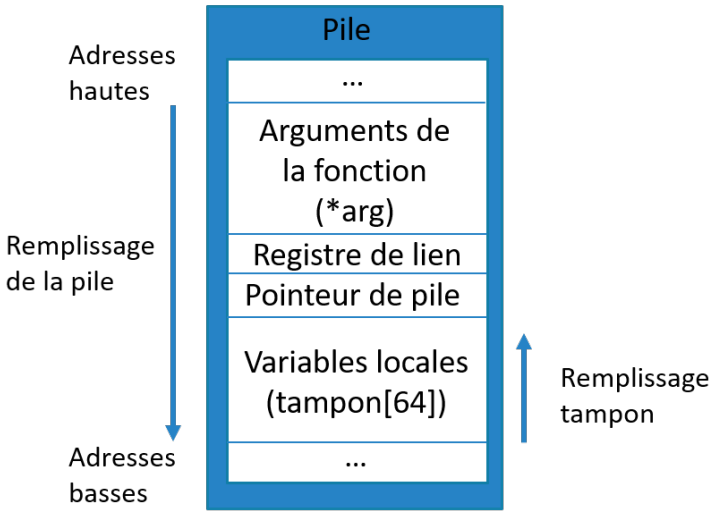

FIGURE 2.4 – *Exemple de pile.*

Un dépassement de tampon peut avoir lieu sur différentes sections d'un programme, les premiers débordements étudiés étaient sur la pile. En effet, en fonction de la section sur laquelle intervient le dépassement, son exploitation sera différente. Dans cette démonstration, nous allons prendre l'exemple avec la pile. Le programme C vulnérable est illustré par la figure 2.5. La pile sert à stocker les données d'exécution du programme, notamment les données associées aux fonctions. À chaque appel de fonction, un *stack frame* est poussé sur la pile. Il contient:

1. **Arguments:** les arguments nécessaires à la fonction qui sont poussés sur la pile. Certains sont directement stockés dans les registres du CPU.
2. **Registre de lien:** un registre qui contient l'adresse de retour une fois la fonction terminée.
3. **Pointeur de pile:** un pointeur pour indiquer l'espace d'adresses réservé sur la pile de la fonction appelante.

```c
#include <stdio.h>
#include <string.h>

void fonction_vuln(char *arg)
{
    char tableau[64];
    strcpy(tableau, arg);
    printf("Copie de arg dans Tableau terminée.\n");
    return;
}

int main(int argc, char *argv[])
{
    if(argc != 2)
        printf("Un seul argument requis.");
    else
        fonction_vuln(argv[1]);
    return 0;
}
```

FIGURE 2.5 – *Exemple de code - Dépassement de tampon.*

4. **Variables locales:** espace réservé pour l'utilisation des variables locales de la fonction.

Il faut noter que le sens de remplissage de la pile diffère de celui du remplissage d'un tampon local. Alors que la pile empile les valeurs des adresses hautes vers les adresses basses, le tampon se remplit des adresses basses vers les adresses hautes. Ce comportement est observé dans la figure 2.4. Pour comprendre pourquoi le remplissage est inversé sur la pile, il faut regarder une deuxième section qui est le tas. Cette section grossit et se remplit inversement à la pile, pour gagner un maximum de place dans la mémoire. Le fait que ces deux sections se remplissent dans un sens inverse permet de réserver un même emplacement pour les deux sections et chacune de ces sections peut prendre la place dont elle a besoin. C'est-à-dire que si la pile a besoin de plus d'espace que le tas ce n'est pas un problème. Si chacune de ces sections avait un emplacement réservé fixe, alors le système pourrait perdre de la mémoire s'il n'utilise pas de mémoire dynamique par exemple (le tas). Cette particularité implique que lors d'un dépassement de tampon dans une variable locale à la fonction, les registres utiles à son fonctionnement peuvent être corrompus.

Pour en revenir à l'exemple, une fonction est appelée (*fonction _vuln*) avec un argument (**arg*). La fonction copie l'argument reçu dans un tampon local nommé *tableau*. Tant que la taille de l'argument est inférieure à la taille du tampon local, ici 64 octets, le programme s'exécutera normalement. Cependant, si un utilisateur malintentionné donne comme argument une chaîne de caractères trop longue, vu qu'aucune taille n'est vérifiée, les arguments sur la pile situés avant le tampon vont être écrasés. De ce fait, le pointeur de pile et le registre de lien qui contiennent les informations pour gérer le retour de la fonction peuvent être écrasés. Comme le registre de lien contient l'adresse qui permet à la fonction de retourner dans la fonction appelante une fois terminée, si un attaquant peut modifier cette adresse, alors il peut potentiellement rediriger le flot

d'exécution du programme. Le comportement est illustré sur la droite de la figure 2.6. À l'aide du dépassement de tampon sur la pile, l'attaquant est capable de venir modifier le registre de lien, et de ce fait, altérer le flot de contrôle du binaire. Cette attaque ouvre donc un champ de possibilités infini pour l'attaquant, car la redirection peut être faite sur n'importe quel code, par exemple une injection de code malicieux.

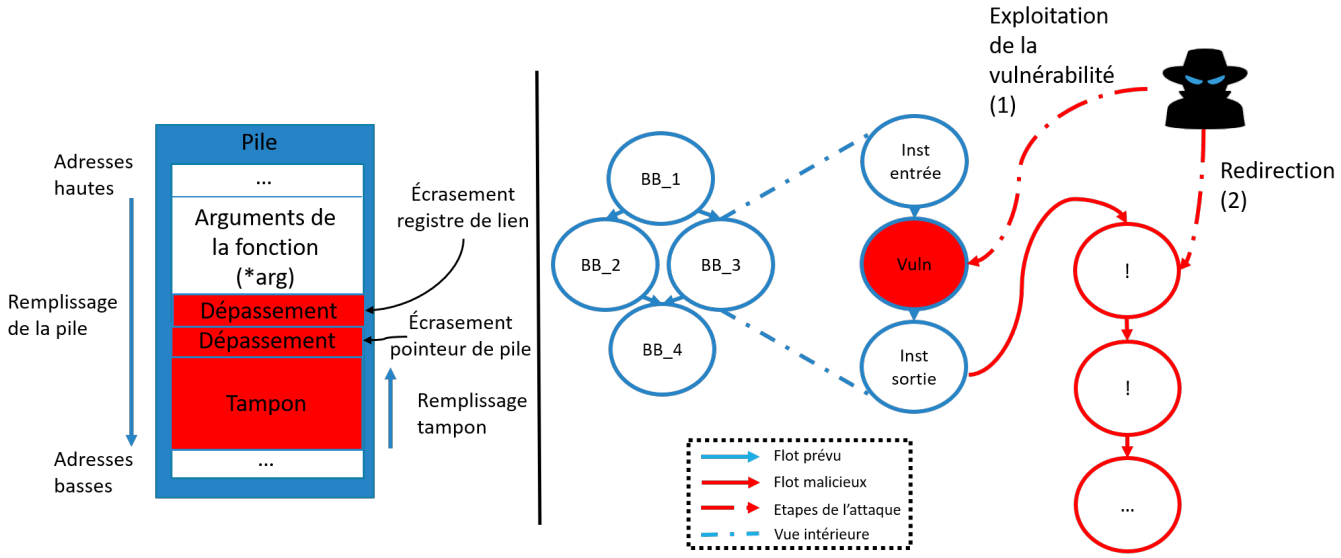

FIGURE 2.6 – *Exemple de dépassement sur la pile.*

2.3 Injection de code

Une attaque par corruption mémoire se passe généralement en deux temps. La première étape est, comme nous venons de le voir, de trouver un moyen d'altérer le flot d'exécution du programme. Pour cela, il faut trouver un moyen de contrôler le compteur de programme (PC) du CPU, c'est-à-dire le registre qui contrôle l'exécution des instructions. Cette étape a été présentée dans la section précédente avec le dépassement de tampon sur la pile. Une fois que le compteur de programme est contrôlé, il faut trouver un moyen d'injecter du code pour faire exécuter une charge malicieuse au CPU, c'est la deuxième étape. Pour cela, il existe deux grandes méthodes: la génération de shellcode ou la réutilisation de code.

2.3.1 Shellcode

Un shellcode [35] est une chaîne de caractères qui représente un code binaire exécutable. L'attaquant crée son code malveillant, le transforme en chaîne de caractères et l'injecte lors de l'attaque. Une possibilité est d'utiliser le tampon pour stocker le shellcode et d'ajouter un rembourrage pour pouvoir d'une part calculer l'adresse du shellcode, mais aussi atteindre l'adresse du registre de lien. La redirection est ensuite très simple, il suffit de faire pointer le registre de lien sur le début du shellcode injecté dans le tampon. L'idée est représentée par la figure 2.7.

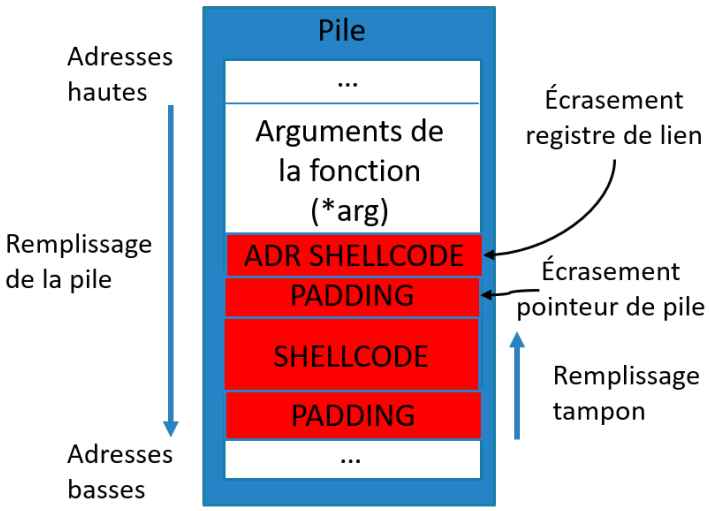

FIGURE 2.7 – *Injection de shellcode.*

Cette méthode est très intéressante, car elle permet de générer n'importe quel programme sous forme de chaîne de caractères. Cependant, bien que la technique soit puissante, elle comporte quelques limitations, car si c'est la première charge d'exploitation, il faut souvent respecter quelques règles:

1. Ne pas contenir d'octet nul.
2. Parfois, il faut respecter un charset alphanumérique.
3. Certaines protections empêchent l'exécution de shellcode. Ce concept est développé dans le chapitre 3.

C'est pourquoi une autre méthode d'attaque a été mise en place: la réutilisation de code [36].

2.3.2 Réutilisation de code

Suite à la mise en place de mécanismes de protection, une nouvelle technique d'injection de code a vu le jour. Au lieu d'injecter du code malveillant dans l'espace d'adressage de l'application à travers un shellcode par exemple, un adversaire peut exploiter le code qui est déjà présent dans l'espace d'adressage et marqué comme exécutable, *i.e.*, la section TEXT. Ces attaques de réutilisation de code ont commencé par des attaques dites de retour en bibliothèque et ont ensuite été généralisées à des attaques de *Return Oriented Programming (ROP)*. Nous décrivons ci-après les concepts techniques de ces deux méthodes.

2.3.2.a Retour en librairie: *Ret2libc*

Les attaques par réutilisation de code qui sont basées sur le principe du retour en librairie altèrent le flot d'exécution d'un programme et le redirigent vers des fonctions

critiques pour la sécurité qui résident dans des librairies partagées ou dans l'exécutable lui-même. La première attaque, qui a utilisé une réutilisation de code, a mis en place un retour sur la librairie. Cet exploit a été présenté par Alexander Peslyak en 1997 [37]. L'idée était d'écraser le registre de lien pour retourner directement sur une fonction critique d'une librairie, dans ce cas, la fonction *system* de la librairie C. À la différence d'une injection de code standard par shellcode, une attaque par *ret2libc* demande seulement la connaissance de l'espace d'adressage des librairies. La figure 2.8 dépeint le déroulement. Un rembourrage est fait jusqu'à atteindre le registre de lien qui permet le contrôle du flot d'exécution. Il suffit ensuite d'injecter l'adresse d'une fonction de la libc, par exemple, *system* est utilisé. Le programme va donc exécuter la fonction *system*, qui utilise les arguments placés au-dessus. Cette méthode peut mener à des exploits plus poussés. Par exemple, Nergal a fait la démonstration de deux techniques, appelées *esp-lifting* et *frame faking*, permettant à un adversaire d'enchaîner les appels aux fonctions lors d'une attaque *ret2libc* [38].

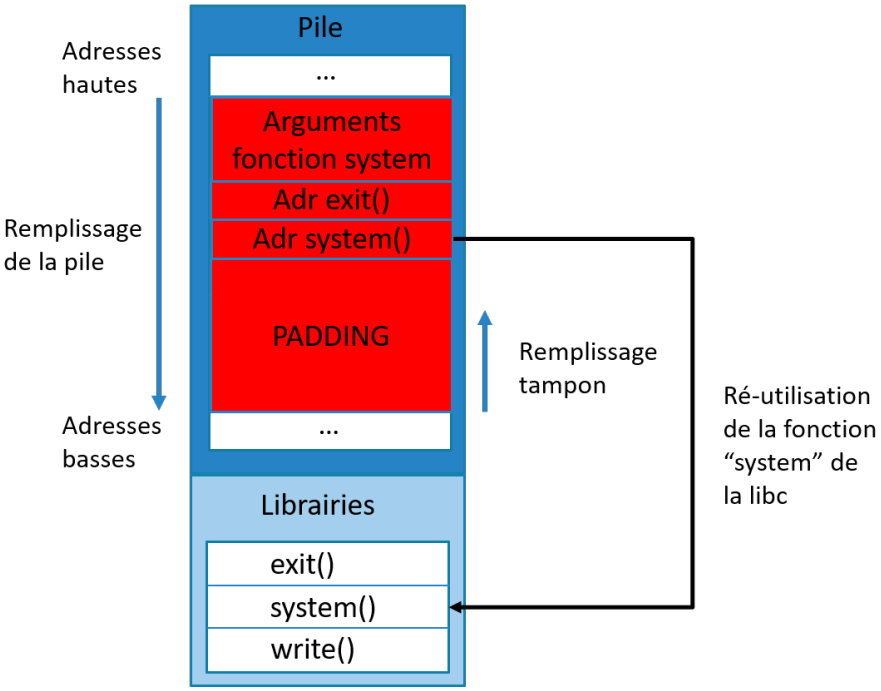

FIGURE 2.8 – *Exemple de réutilisation de code: ret2libc.*

Bien que la méthode présente l'avantage de simplicité d'utilisation, car elle repose principalement sur la librairie C, elle contient quelques contraintes:

1. Il faut que le programme utilise une librairie.
2. Il faut connaître l'espace d'adressage de la librairie C.
3. Il faut connaître la version de la librairie utilisée.
4. Il faut que les appels systèmes critiques ne soient pas bloqués, ou supprimés de la librairie.

Toutes ces contraintes font que la réutilisation de code par retour en librairie n'est pas forcément applicable dans tous les cas d'usages. De ces limitations sont nées des

méthodes de réutilisation de codes plus complexes, mais aussi plus complètes. La plus connue de ces méthodes est le *Return Oriented Programming* (ROP).

2.3.2.b Return Oriented Programming (ROP)

Les attaques par réutilisation de code qui sont basées sur le principe sur le *Return Oriented Programming* (ROP) [39] combinent et exécutent une chaîne de courtes séquences d'instructions qui sont dispersées dans l'espace d'adressage de la section *Text* d'une application. Chaque séquence se termine par une instruction indirecte de branchement ou une instruction de retour pour transférer le contrôle d'une séquence à la séquence suivante. Ces séquences sont appelées des *gadgets*. La programmation à l'aide de ces gadgets s'avère Turing-complet, c'est-à-dire que la combinaison de ces gadgets permet de programmer tout comportement possible. On peut voir la programmation orientée retour comme une programmation assembleur utilisant des gadgets comme instructions.

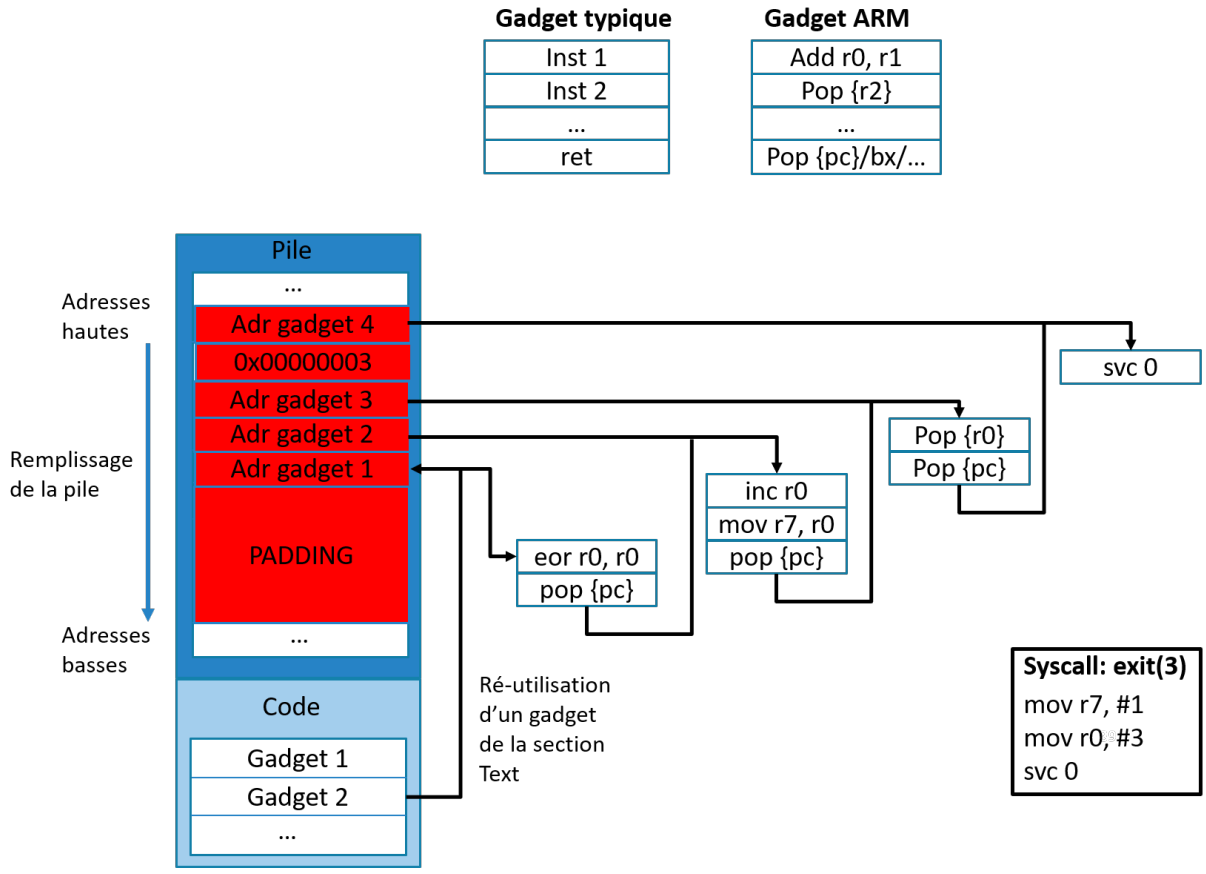

FIGURE 2.9 – *Exemple de réutilisation de code: ROP.*

La figure 2.9 illustre le fonctionnement. Dans un premier temps, il faut identifier les gadgets nécessaires à l'élaboration du programme malicieux. Dans cet exemple, nous avons identifié quatre gadgets ARM. En reprenant l'exemple de dépassement de tampon sur la pile, l'idée est de venir écraser le registre de lien avec l'adresse du premier gadget. La première instruction exécutée, *eor r0,r0*, réalise une opération XOR sur le registre *r0* qui a pour effet de le mettre à 0. La dernière instruction d'un gadget

est une instruction indirecte de branchement, typiquement un *ret* ou plutôt un *pop {pc}* sur une architecture ARM. Le programme exécute alors le prochain gadget sur la pile (gadget 2). La valeur de *r0* est incrémentée de 1, et la valeur 1 est mise dans *r7* pour contrôler le numéro de l'appel système. Ici il correspond à *exit()*. Le troisième gadget récupère un argument sur la pile (la valeur 3) et la place dans le registre *r0* puis passe à l'exécution du gadget 4. Ce dernier exécute l'instruction *svc 0* qui déclenche l'appel système réglé dans le registre *r7* (*exit(3)*). Il est donc possible de facilement chaîner plusieurs gadgets pour construire un programme plus élaboré et cette chaîne de gadget est communément appelée "*ropchain*".

La mise en place de ROP a été démontrée sur plusieurs architectures, incluant ARM [40], x86 [41], et Atmel AVR [42]. Ces injections de code sont très utilisées dans les attaques récentes et plusieurs outils ont été développés pour faciliter l'identification de gadget et la composition de ropchain. Les plus connus sont ROPgadget [43] et Ropper [44]. De plus Bosman *et al.* ont prouvé qu'il est possible d'utiliser les structures des signaux dans un système d'exploitation pour faciliter et rendre plus portable la création de *ropchain* [45]. Bittau *et al.* [46] ont montré qu'il est possible d'exploiter un programme en injectant une *ropchain* sans connaissance du binaire au préalable. Toute l'exploitation et l'identification des gadgets se passent à l'aveugle. La même méthode existe à base d'instruction de saut au lieu d'instruction de retour. Cette méthode est appelée *Jump Oriented Programming* (JOP) [47]. Ces différents outils et techniques font des attaques par réutilisation de code un vrai fléau qui arrive à contourner nombre de protections.

2.4 Différentes attaques

Dans la section précédente, nous avons vu plusieurs méthodes d'injection de code. Nous avons aussi vu un type d'exploitation mémoire qui est le dépassement de tampon sur la pile. Cependant, il existe énormément de moyens différents pour exploiter le flot de contrôle d'un programme. Récemment, Microsoft a publié une étude sur la proportion de types d'attaques présente au sein de leur entreprise [27]. La Figure 2.10 en est un extrait. Le graphique illustre la répartition des types de corruption mémoire à travers le temps, de 2006 à 2018. Ce graphique est très intéressant, d'une part on peut observer que le nombre de *Common Vulnerabilities & Exposures* (CVE) a explosé dans le temps. On passe de 134 vulnérabilités en 2006 à plus de 600 en 2018. D'autre part, on voit que certains types de corruptions étaient très présents en 2006, le dépassement de tampon sur la pile par exemple, alors que cette catégorie a quasiment disparu en 2018. On peut expliquer cette disparition par les différentes protections déployées, mais aussi une large communication sur ce type de vulnérabilité. Au contraire, d'autres vulnérabilités sont devenues beaucoup plus présentes, c'est le cas des confusions de type et des utilisations de "variables non initialisées". Même au sein de chacune de ces catégories, il existe de nombreuses méthodes d'exploitation. Nous n'allons pas pouvoir couvrir tout le panel d'attaques possibles parce qu'il existe de nombreuses variantes. Cependant, il est important de comprendre les bases des catégories les plus connues pour avoir une meilleure vue d'ensemble et d'ensuite proposer des solutions de contremesures pertinentes.

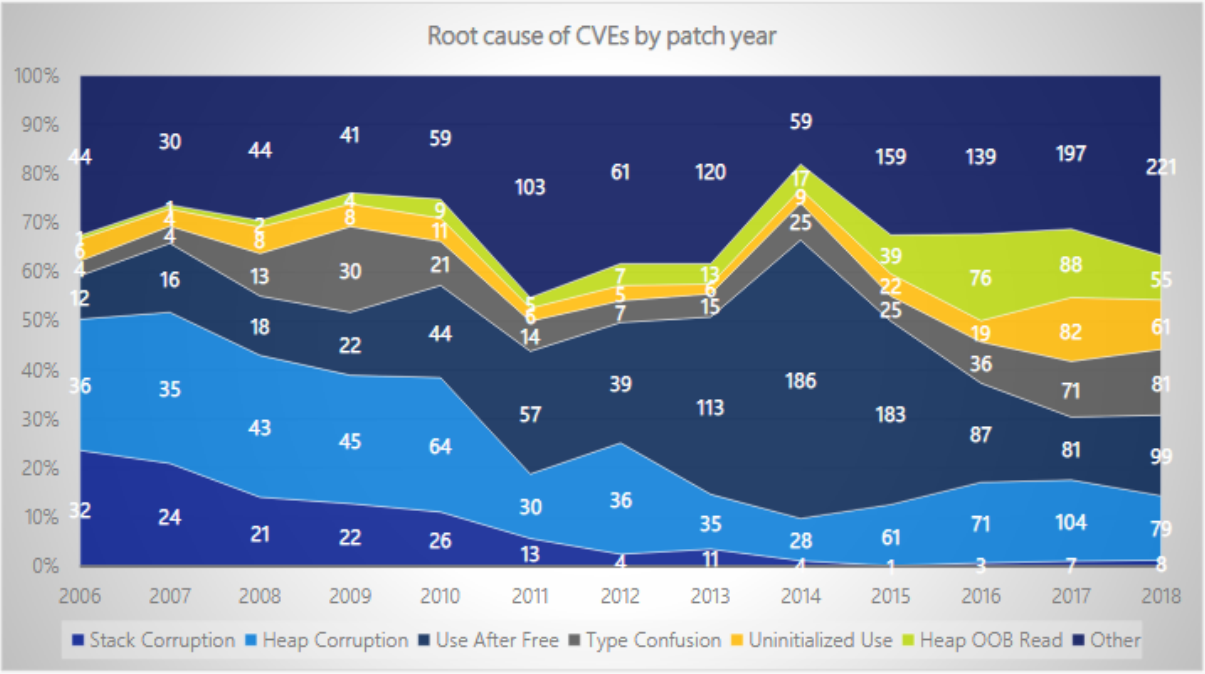

FIGURE 2.10 – *Répartition des bugs de corruption de mémoire chez Microsoft.*

2.4.1 Variable non initialisée

La vulnérabilité reposant sur une variable non initialisée arrive lorsque le développeur déclare une variable dans son code, mais oublie de l'initialiser. Cette vulnérabilité est définie dans le Common Weakness Enumeration (CWE) [48]. À l'exécution, la variable a une valeur non déterminée, et il est impossible de prédire cette valeur ou même le comportement du programme. Cependant, cette vulnérabilité peut être exploitée par un attaquant pour récupérer le contrôle de l'application. Plusieurs articles et travaux expliquent comment exploiter ce genre de bug. Par exemple, Ratul Gupta donne un exemple concret d'exploitation d'une variable non initialisée sur la pile pour exploiter le noyau de Windows [49]. Halvar Flake explique plus généralement le flot d'attaque et de reconnaissance à tenir pour exploiter ce type de bug [50]. Cette vulnérabilité, peu présente dans les années 2006 à 2014, voit arriver beaucoup plus de cas à partir de 2017 et devient une vraie menace.

2.4.2 Confusion de type

La confusion de type est une erreur typique en programmation. Le développeur alloue ou initialise une ressource comme un pointeur, un objet ou une variable avec un type bien défini. Lors de l'exécution, un accès est fait sur cette ressource, mais en utilisant un type incompatible avec celui défini précédemment. Cette vulnérabilité est définie dans le CWE [51]. Lorsque le programme accède à une ressource en utilisant un type incompatible, cela peut déclencher des erreurs logiques, car la ressource n'a pas les propriétés attendues. Dans les langages sans sécurité mémoire, tels que le C et le C++, une confusion de type peut entraîner un accès mémoire hors limites et une exécution de code arbitraire. Bien que cette faiblesse soit souvent associée à des unions

lors de l'analyse de données avec de nombreux types d'objets intégrés différents en C, elle peut être présente dans toute application pouvant interpréter la même variable ou le même emplacement mémoire de multiples façons. De la même manière que pour le cas des variables non initialisées, les vulnérabilités de confusion de type n'étaient pas très présentes des années 2006 à 2014. Leur nombre s'est vu accru à partir de 2017. Comme expliquée lors du SSTIC [52], même si ce n'est pas la vulnérabilité la plus répandue, sa nature fait que les protections actuelles ne sont pas capables de la contrer ou même de la détecter. Dans un article publié par Microsoft, Jeong Wook explique comment fonctionne un bug de confusion de type et comment l'exploiter [53].

2.4.3 Lecture de la mémoire hors limite

Une lecture de la mémoire hors limite veut dire que le programme lit des données après la fin, ou avant le début, de la mémoire tampon prévue. Cette vulnérabilité est définie dans le CWE [54]. Un attaquant peut donc lire des données sensibles ou provoquer un plantage. Un plantage peut se produire lorsque le code lit les données jusqu'à rencontrer un octet nul par exemple. S'il n'y a pas d'octet nul et que le pointeur de lecture arrive hors limite de la mémoire, alors une erreur de segmentation se produit. Cette vulnérabilité peut aussi être utilisée pour faire fuiter des données utiles à l'exploitation d'une autre faille comme un dépassement de tampon. En effet, comme l'attaquant peut lire des données de mémoire sensible, il est potentiellement capable de lire l'adresse de base de l'ASLR ou de lire le canari (ces concepts sont expliqués dans le chapitre 3).

2.4.4 Utilisation après libération de mémoire (UAF)

Le référencement de la mémoire après sa libération peut provoquer le plantage d'un programme, l'utilisation de valeurs inattendues ou l'exécution de code arbitraire. Cette vulnérabilité est définie dans le CWE [55]. L'utilisation d'un espace mémoire préalablement libéré peut avoir un certain nombre de conséquences négatives, allant de la corruption de données valides à l'exécution d'un code arbitraire. La manière la plus simple de corrompre des données consiste à réutiliser la mémoire libérée par le système. Par exemple, la mémoire en question est attribuée à un autre pointeur après avoir été libérée. Le pointeur original vers la mémoire libérée est utilisé à nouveau et pointe vers une nouvelle allocation. Lorsque les données sont modifiées, elles corrompent alors la mémoire utilisée, ce qui induit un comportement indéfini dans le processus. Cette méthode est bien expliquée dans un blogpost de Romain Bentz [56]. Depuis les années 2006, les UAF occupent quasiment 30% de toutes les failles par corruption mémoire chez Microsoft et sont largement exploités dans tous les logiciels courants comme Google Chrome, Linux, ou encore Android. Bien que la proportion ne change pas, leur nombre augmente de façon significative avec les années. Il est possible de trouver des exemples de CVE sur les UAF dans des articles. Récemment, l'équipe du projet zéro de Google a divulgué les détails d'un bug UAF dans Chrome [57]. En 2015, Wen Xu *et al.* ont expliqué comment exploiter un UAF dans le noyau Linux [58].

2.4.5 Corruption du tas

Les corruptions du tas peuvent prendre différentes formes. Les vulnérabilités UAF sont un sous-ensemble de corruption du tas. Mais il existe nombreux autres sous-ensembles : un site maintenu par la communauté de hacking référence la majorité d'entre eux [59]. L'équipe de CTF shellphish a aussi mis à disposition d'excellentes ressources sur l'exploitation du tas avec les différentes vulnérabilités et des exemples d'exploitation [60]. L'important à retenir est qu'il existe énormément de bugs liés à la programmation avec des langages permissifs tels que le C. De ces bugs découlent autant d'exploitations possibles, avec des techniques différentes, qui rendent les protections très compliquées à mettre en place.

Pour conclure, quelle que soit la technique d'exploitation du flot d'exécution d'un programme, le résultat reste le même: l'attaquant est capable de contrôler le comportement du programme. Nous avons vu qu'en plus de toutes ces possibilités d'exploitation de vulnérabilité, il y a aussi différentes méthodes d'injection de code. Tous ces choix laissent à l'attaquant un large champ de possibilités pour leur exploitation.

2.5 Des attaques en deux temps

Au final, une attaque par corruption de mémoire consiste en la première phase d'exploitation d'un système. Ces attaques permettent à un utilisateur malintentionné de prendre contrôle d'un dispositif, et de lui laisser un large choix d'applications. Cette deuxième phase est la phase post-exploitation. La figure 2.11 résume le déroulement complet.

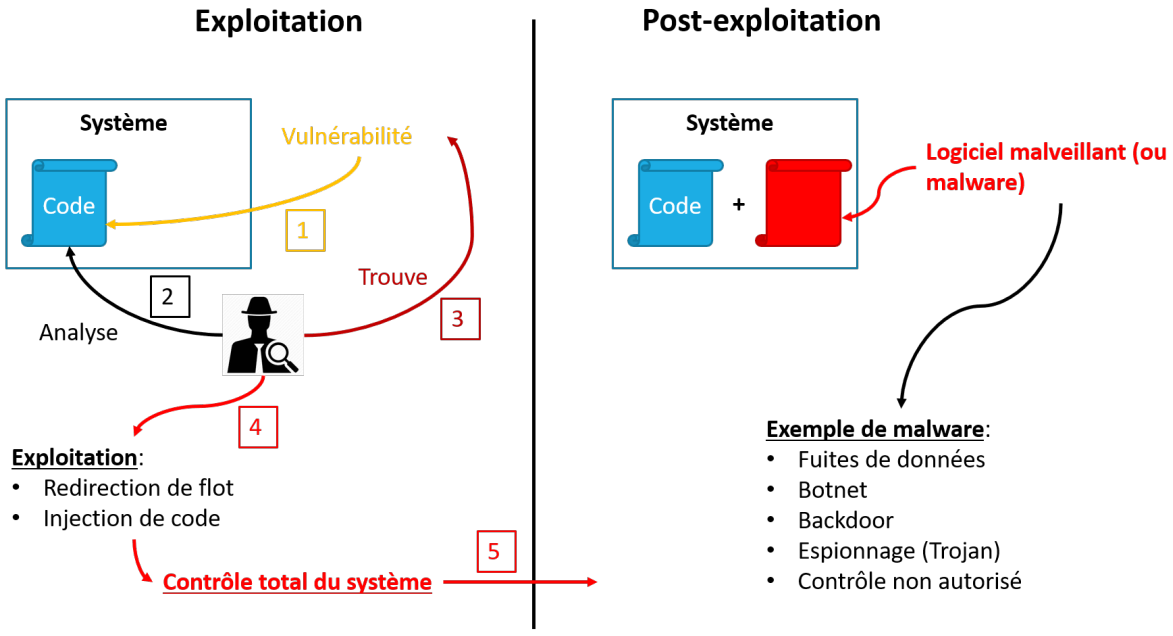

FIGURE 2.11 – *Attaque en deux temps.*

Un système est défini, exécutant un programme, avec un code contenant une vul-

nérabilité (1). Un utilisateur malveillant analyse ce programme (2), et trouve la vulnérabilité (3). À partir de cet instant, il est en mesure d'exploiter cette vulnérabilité (4). L'exploitation se passe en deux étapes comme nous venons de le voir; une étape pour rediriger de flot de contrôle du programme et une étape d'injection de code pour obtenir un contrôle total du système (5). Généralement, l'étape d'exploitation vise seulement à prendre contrôle du système, à travers une charge malicieuse qui corrompt le fonctionnement normal du programme. C'est après que l'attaque réelle commence, on l'appelle l'étape de post-exploitation. Elle consiste à tirer parti du contrôle du système pour exécuter un nouveau programme, généralement malveillant, aussi connu sous le nom de *malware*. Nombreux exemples existent :

- Des fuites de bases de données pour obtenir e-mails et mots de passe.
- La mise en place de Botnet pour avoir un réseau de machines fantômes et augmenter sa puissance de calcul ou de frappe DDOS.
- La mise en place de portes dérobées (*backdoor*) pour entrer facilement dans le système à posteriori.
- L'intrusion d'un logiciel espion (*Trojan*) pour garder des informations sur le fonctionnement du système ou d'une entreprise.
- Un contrôle non autorisé des fonctionnalités comme l'éclairage public ou une chaîne de production industrielle.

Les applications sont diverses, variées et limitées à l'imagination de l'attaquant. Cela rend la partie post-exploitation très compliquée à détecter ou à prévenir. Entre ces possibilités d'applications multiples et le nombre considérable de méthodes d'injection de code et de redirection de flot, nombreuses sont les protections qui ont été déployées (que nous détaillons dans le chapitre 3). Pour bien comprendre cet ensemble (attaques et contremesures), et ainsi proposer des solutions de protection optimales, nous avons élaboré une plateforme.

2.6 Plateforme de développement et de test d'attaques

Pour pouvoir tester et mieux comprendre les différents types d'attaques existants, mais aussi les différents types d'injection de code, nous avons défini une plateforme. L'idée est de tester ces fonctionnalités sur ce qui va le plus s'approcher d'un système final IdO, et donc d'éviter de faire cette plateforme sur un PC avec une architecture x86. Un système final IdO consiste en :

- Une partie Cloud, souvent représentée par un serveur géré par un opérateur.
- Une partie embarquée, représentant différents objets connectés qui peuvent agir comme des actionneurs ou des capteurs.

- Une partie réseau, qui permet au Cloud et aux objets connectés de communiquer ensemble.

Dans cette thèse, nous travaillons principalement sur la partie embarquée. Les passerelles, qui sont finalement des objets embarqués permettant la communication entre le Cloud et les objets connectés, sont un point crucial dans l'écosystème IdO. C'est sur ce type de système que nous avons décidé de tester nos travaux.

2.6.1 Développement de la passerelle

Avant de développer la passerelle, nous avons défini un écosystème IdO pour mettre en place la plateforme (figure 2.12). On retrouve un réseau local et un réseau étendu qui correspond à Internet. Il y a des capteurs et des actionneurs, qui permettent d'interagir avec l'environnement. Un PC principal sert d'administrateur de cet écosystème et les passerelles sont présentes pour faire le lien entre l'administrateur et les éléments d'interaction. Enfin, un attaquant peut accéder au réseau pour venir essayer de compromettre les passerelles et prendre le contrôle des capteurs et des actionneurs.

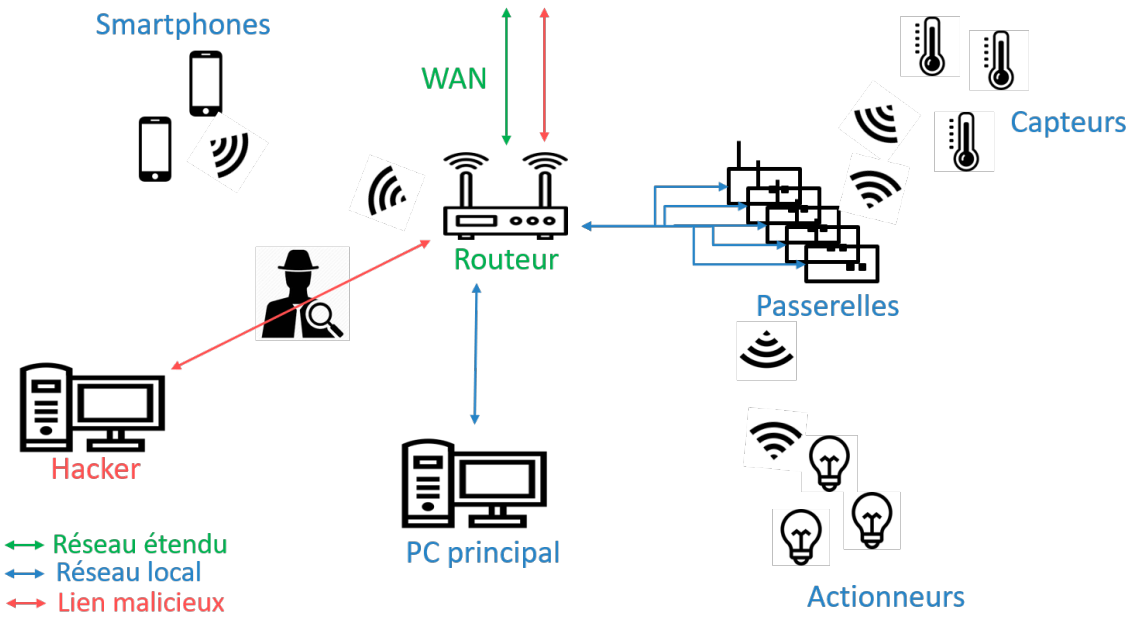

FIGURE 2.12 – *Architecture de la plateforme.*

La passerelle est au coeur de notre sujet et du modèle de menace, car étant un point central de toute communication, elle possède une grande surface d'attaques. Pour choisir sur quel environnement déployer la passerelle, nous avons posé un cahier des charges qui permet d'une part d'avoir un environnement réaliste pour tester les différentes attaques et d'autre part un environnement pour tester les protections. L'idée est de tirer profit d'un maximum des fonctionnalités matérielles et logicielles d'un système embarqué pour mettre en place la meilleure sécurité possible. Pour cela, il faut un système avec une grande marge de manoeuvre sur la configuration, aussi bien niveau matériel que logiciel. Pour le matériel, les architectures ARM sont très utilisées dans le milieu de l'embarqué et de l'IdO. C'est donc naturellement que nous nous

sommes dirigés vers un processeur ARM 32 bit. De plus, dans le milieu de la sécurité, nombreuses sont les solutions qui utilisent les FPGA pour prototyper les solutions matérielles. Notre choix s'est donc porté sur un processeur hybride, *i.e.*, plusieurs CPU ARM 32 bits interfacés avec un FPGA. La carte de développement Zybo [61] répond à tous ces critères. Elle possède des actionneurs comme des switchs et des boutons-poussoirs, un emplacement carte SD pour facilement mettre un OS comme Linux, et des LED pour communiquer avec l'utilisateur. Les mémoires intégrées, ainsi que l'interface USB et Ethernet, simplifient au maximum la phase de conception et de test. De plus, la carte Zybo possède un processeur hybride : le Zynq-7000 [62].

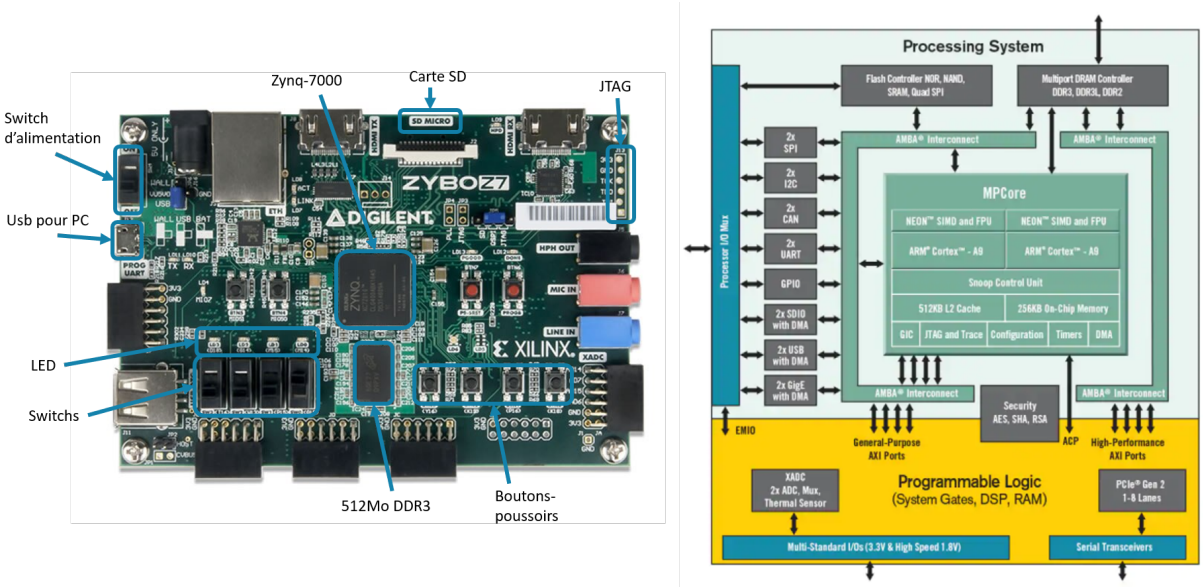

FIGURE 2.13 – *Illustration Zybo et Zynq-7000.*

Le Zynq-7000 (à droite sur la figure 2.13) possède deux CPU (ARM cortex-a9 [63] 667 MHz), communiquant avec un FPGA à travers un bus AXI. On retrouve les principales interfaces de communication, à savoir des bus SPI, I2C, UART, ou encore des GPIO. De plus, il est possible de configurer les accès (rapidité ou droits par exemple) entre le FPGA et les CPU, les différentes interruptions requises (à travers le GIC), ou encore des sondes de débogage présentes dans la partie *Processing System*. Pour implémenter les couches logicielles, l'utilisation de Yocto [64] est choisie. Yocto est un projet qui permet de créer des distributions Linux personnalisées pour n'importe quel processeur/environnement. Il permet donc une grande marge de manoeuvre dans la gestion du système d'exploitation et de ses versions, des noyaux utilisés, des drivers ou encore des applications. La mise en place d'un environnement pour tester des attaques et des protections s'en retrouve plus modulable, bien que cela demande plus de travail en amont. À l'aide de cette plateforme, nous avons pu tester et mieux comprendre toutes les attaques présentées ci-dessus, comprendre pourquoi les protections actuelles sont limitées, et avoir des pistes de contributions. De plus, cette plateforme sera réutilisée dans le cadre de nos travaux sur les différentes contributions que nous proposons. Une fois que les attaques ont été maitrisées, nous avons décidé de faire un démonstrateur, pour étudier la criticité des vulnérabilités et la facilité de prise de contrôle d'un système.

2.6.2 Démonstrateur

Pour cette démonstration, nous avons mis en oeuvre un scénario basique de l'Internet des Objets. L'idée est de montrer qu'une fois une vulnérabilité trouvée dans un produit, il est très facile de prendre contrôle de ce dernier, et de bénéficier de ses privilèges. Pour cela, nous utilisons un scénario simple, basé sur l'utilisation du protocole LwM2M [26].

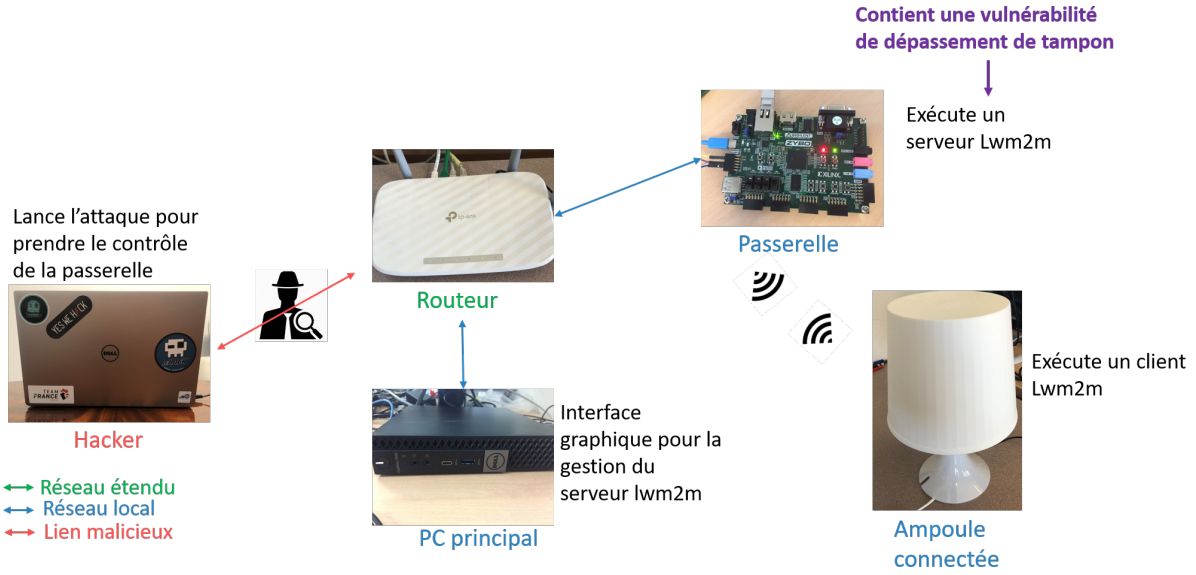

FIGURE 2.14 – *Démonstrateur.*

La figure 2.14 illustre l'organisation de notre démonstrateur. La passerelle exécute un serveur LwM2M alors que l'ampoule connectée exécute un client LwM2M. En d'autres termes, le PC principal s'occupe de gérer l'ampoule connectée à travers une interface graphique qui permet de gérer le serveur de la passerelle. La passerelle quant à elle s'occupe de transmettre les ordres finaux à l'ampoule connectée. Nous avons au préalable placé une vulnérabilité dans le serveur LwM2M qui s'exécute sur la passerelle.

La figure 2.15 explique plus en détail la démonstration. Le serveur LwM2M, exécuté sur la carte Zybo, communique avec le PC principal et l'interface graphique à travers une websocket. Ici, le scénario de l'attaquant est de prendre le contrôle de la passerelle, et de ce fait du serveur LwM2M, pour éteindre l'ampoule. L'attaquant exploite une vulnérabilité de dépassement de tampon sur le serveur à travers une communication UDP. L'attaquant formate un paquet réseau pour s'enregistrer sur le serveur comme étant un objet, et profite d'un dépassement de tampon pour écraser la sauvegarde du registre de lien. Il injecte ensuite une *ropchain* pour exécuter un *shell* et ainsi contourner les protections ASLR et DEP. Comme le CPU de la Zybo fonctionne en 32 bits, l'attaque utilise un bruteforce du *StackGuard*. Ainsi, après plusieurs tentatives, l'attaquant prend contrôle du serveur, et clôture la partie exploitation. La partie post-exploitation de l'attaque peut donc commencer: prendre le contrôle de l'ampoule. Comme l'attaquant possède un accès direct sur la passerelle, il peut exécuter tout type de commande. Par exemple, envoyer un ordre d'écriture LwM2M sur l'objet ampoule pour l'éteindre. Cette démonstration très visuelle montre l'impact que peut avoir un logiciel vulnérable dans

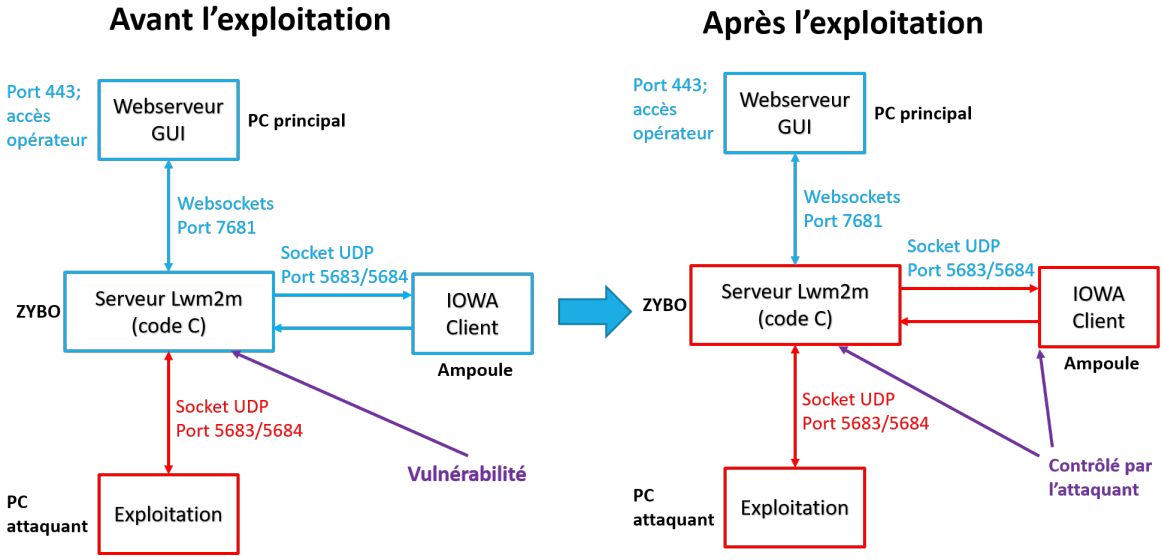

FIGURE 2.15 – *Détails de la démonstration.*

un bâtiment connecté par exemple. Elle permet de voir avec quelles rapidité et facilité, un utilisateur malveillant peut prendre le contrôle d'un système et ainsi compromettre des éléments d'un réseau. De plus, la démonstration démontre que même avec des protections, *i.e.*, *Data Execution Prevention (DEP), StackGuard, Address Space Layout Randomization (ASLR)*, visant à empêcher les corruptions de mémoire, il est toujours relativement facile d'exploiter ces vulnérabilités.

2.6.3 Valorisation

De notre point de vue, l'apprentissage des méthodes d'attaque est une étape obligatoire pour pouvoir proposer des méthodes de protection pertinentes. Cette partie demande beaucoup d'investissement et n'est pas simple à valoriser à travers des publications. Les attaques étudiées sont connues dans le domaine, et rien de nouveau n'a été apporté de ce côté-là; les attaques connues dans l'état de l'art suffisent déjà à contourner les protections actuelles (que nous expliquons dans le détail par la suite). Il existe des évènements de sécurité au cours desquels se déroulent des concours de hacking, plus communément appelés *Capture The Flag* (CTF). Ces évènements se déroulent seuls ou en équipe et permettent de tester ses connaissances dans le domaine. On trouve plusieurs catégories, du web, de la cryptographie, du forensique, de la rétro-ingénierie et plus particulièrement ce qui nous intéresse dans cette thèse, de l'exploitation de binaire. Au cours de la thèse, j'ai pu participer à nombre de CTF, en ligne ou sur site, seul ou en équipe, notamment :

1. Qualification European Cyber Week (ECW) 2018 [65] seul : 4/ 600.
2. Pré-qualification European CyberSecurity Challenge (ECSC) 2019 [66] seul: 12/ 1200.
3. Qualification European CyberSecurity Challenge (ECSC) 2019 seul: 6/50.
4. Qualification European Cyber Week (ECW) 2019 seul : 1/ 600.

5. Finale European Cyber Week (ECW) 2019 en équipe : 1/12.

L'ECW est la plus grosse compétition française de cybersécurité étudiante, organisée par le pôle cyber-excellence de Rennes, Thales et Airbus. J'ai pu participer en 2018 et remporter l'édition de 2019 en terminant 1er des qualifications et de la finale. L'ECSC est un challenge européen organisé par l'ENISA. Chaque pays participant doit proposer une équipe de 10 joueurs de moins de 25 ans. En France, c'est l'ANSSI qui a pour mission de constituer cette équipe. Ce classement lors des préqualifications m'a permis de participer aux qualifications finales. Une fois encore, le classement aux qualifications finales m'a permis d'intégrer l'équipe française de cybersécurité en 2019, et de participer à une semaine d'entraînement à Paris avec des coachs de l'ANSSI et de représenter la France lors de la finale de l'ECSC à Bucarest en Roumanie. Ces aventures ont permis de mettre en pratique et de valoriser toutes les connaissances acquises durant la phase d'apprentissage de l'axe "attaque" de la thèse. À savoir, la mise en place de techniques de rétro-ingénierie qui visent à analyser un code binaire pour contourner des mécanismes de protection et rechercher des vulnérabilités, afin de les exploiter (redirection de flot et injection de code) pour prendre le contrôle d'un système.

2.7 Synthèse

À travers ce chapitre, nous avons mieux compris le modèle de menace et les attaques qui en découlent. Lorsqu'un programmeur utilise des langages de programmation dits "permissifs", c'est-à-dire que ces langages autorisent une gestion fine de la mémoire, il est possible de perdre le contrôle de son système à partir d'une simple erreur de programmation. Nous avons démontré ce cas à partir d'un exemple de dépassement de tampon sur la pile et montré qu'une fois que l'attaquant contrôle le pointeur d'instruction, il dispose de plusieurs méthodes pour injecter du code qui sont dépendantes des contraintes rencontrées. Nous avons ensuite précisé les différentes méthodes de corruption de mémoire pour contrôler le pointeur d'instruction. Nous avons expliqué que suite à cette étape d'exploitation, l'attaquant déploie sa vraie attaque: c'est l'étape post-exploitation. L'étude et la prise de connaissance de ces méthodes nous a permis de mieux comprendre pourquoi il est difficile de détecter la corruption en amont et de développer une vraie expertise dans l'implémentation d'attaques de corruption de mémoire. Cela nous a aussi permis de construire une plateforme pour tester ces attaques, qui sera utilisée pour tester nos contributions. Cette expertise nous a aussi permis de participer à plusieurs compétitions de sécurité dite "CTF". Nous avons gagné de prestigieuses compétitions comme l'ECW et fait parti de l'équipe de France de sécurité en 2019 lors de l'ECSC. Cette valorisation et l'apprentissage mené au cours de ces travaux nous ont donné les clés pour comprendre les protections actuelles et leurs limitations, que nous explorons dans le chapitre suivant.

III

État de l'art

Sommaire

3.1 Motivations

En réponse à l'ampleur de ces attaques, de nombreuses protections ont été définies et étudiées. Maintenant que nous avons compris et analysé comment fonctionnent les attaques par corruption de mémoire, nous pouvons passer à l'axe 3 de la thèse (figure 3.1): "Compréhension et maîtrise des méthodes de protection à l'état de l'art".

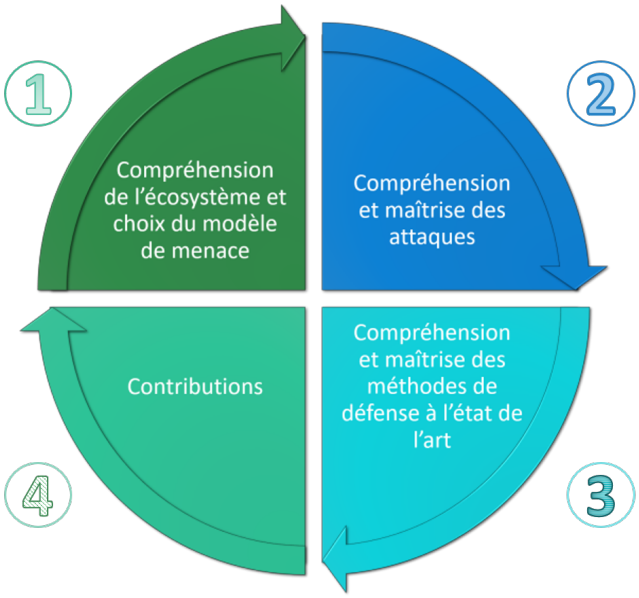

FIGURE 3.1 – *Axes de la thèse*

Ce chapitre a pour objectif de fournir un contexte détaillé sur l'état de l'art relatif aux mécanismes de défense et de détection des vulnérabilités de corruption de mémoire. De nombreuses méthodes existent, mais aucune classification n'est proposée dans la littérature permettant de mieux aborder le sujet. Nous nous inspirons alors de la classification proposée par Jean Arlat sur la tolérance aux fautes [67] pour proposer une classification des méthodes de protection et plus généralement de la sécurité d'un système. Cette dernière peut être évaluée selon quatre grands axes (que l'on peut retrouver dans la figure 3.2) :

- **Fournir un système sécurisé** : l'aptitude à délivrer un service de confiance.
- **Analyse de la sécurité en cours de fonctionnement** : assurer que les attaques puissent être détectées pendant l'exécution du système.
- **Évitement des attaques** : prévenir les attaques en éliminant un maximum de vulnérabilités.
- **Acceptation des attaques** : en faisant l'hypothèse que l'on ne peut empêcher 100% des attaques, réussir à retrouver un état sûr et connu même après une corruption.

Pour parvenir à une sécurité optimale, il est nécessaire de combiner un ensemble de méthodes, qui peuvent être classées selon deux grandes familles :

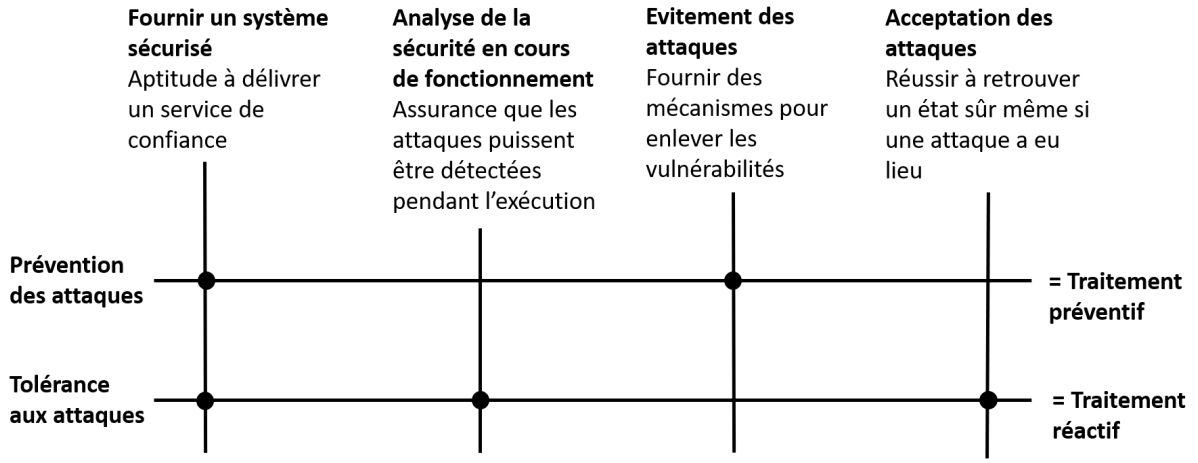

FIGURE 3.2 – *Groupements des moyens pour la sécurité d'un système.*

- **La prévention des attaques** : consiste à empêcher des attaques et supprimer des vulnérabilités. Ces méthodes interviennent principalement suivant deux axes: sécuriser le système et éviter des attaques. Le but étant de traiter le système de manière préventive et passive, pour éviter un maximum d'attaques.
- **La tolérance aux attaques** : consiste à détecter des attaques et rétablir le système. Ici, l'idée est d'agir de manière réactive pour détecter une attaque et proposer une contremesure pour rétablir le système.

Dans ce chapitre, nous allons détailler ces deux grandes familles pour comprendre les méthodes de protection actuelles ainsi que leurs limitations.

3.2 Prévention des attaques

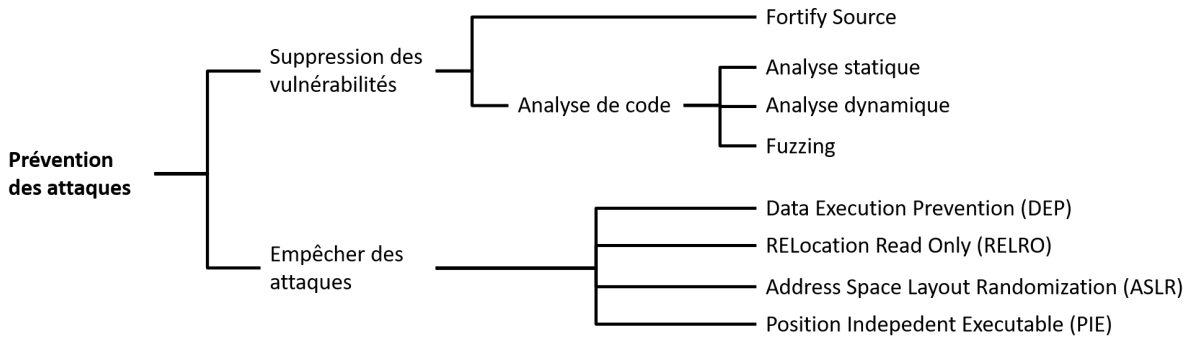

FIGURE 3.3 – *Classification des méthodes de prévention des attaques.*

Nous proposons de classer les méthodes de prévention selon deux axes (voir figure 3.3):

- **Suppression des vulnérabilités** : concerne les méthodes qui vont directement supprimer les vulnérabilités du programme.

- **Empêcher les attaques** : concerne les méthodes qui vont empêcher certaines catégories d'attaques, même si une vulnérabilité est présente.

Ces protections peuvent être introduites à différents niveaux. Certaines sont mises en place par le compilateur, d'autres sont intégrées directement par le système d'exploitation, alors que certaines doivent être fournies par le développeur. Dans les sections suivantes, nous allons voir quels sont les bénéfices apportés par chaque élément, ainsi que les limitations.

3.2.1 Suppression des vulnérabilités

Dans un premier temps, nous détaillons les mécanismes et méthodes existantes visant à éliminer les vulnérabilités d'un programme.

3.2.1.a Fortify Source

La protection FORTIFY SOURCE est une fonctionnalité proposée par GCC. Elle entraîne des vérifications légères pour détecter certaines erreurs de dépassement de tampon lors de l'utilisation de diverses fonctions de manipulation de chaînes et de mémoire (par exemple, *memcpy*, *memset*, *stpcpy*, *strcpy*, *strncpy*, *strcatv*, *strncat*, *sprintf*, *snprintf*, *vsprintf*, *vsnprintf*, *gets*). Pour certaines fonctions, la cohérence des arguments est vérifiée; par exemple, on vérifie que *open* a été fourni avec un argument *mode* lorsque les indicateurs spécifiés incluent *O_CREAT*. Dans d'autres cas, c'est la taille des arguments qui est vérifiée. Il existe deux niveaux de sécurité pour le FORTIFY SOURCE:

1. Le niveau 1, qui inclut des vérifications qui ne changent pas le comportement des programmes. C'est-à-dire que toutes les vérifications sont faites à la compilation et aucun code n'est ajouté dans le binaire final. Les erreurs relevées sont généralement indiquées par des avertissements du compilateur.

2. Le niveau 2, qui propose une vérification supplémentaire, peut changer le comportement des programmes en ajoutant des vérifications à l'exécution. De ce fait, il peut provoquer des erreurs à l'exécution s'il détecte un problème, mais aussi rendre le code moins stable.

Globalement, le niveau 2 de FORTIFY SOURCE est plus sûr, mais propose une stratégie de compilation légèrement plus risquée. Il faut s'assurer d'avoir des tests de régression très forts pour le code compilé pour prouver que le compilateur n'a introduit aucun comportement inattendu. De plus, cette solution ne relève pas tous les problèmes, seulement certains cas courants.

3.2.1.b Analyse de code

L'analyse de code source ou compilé, qu'elle soit statique ou dynamique, consiste en une première ligne de défense contre les vulnérabilités. L'analyse statique consiste à étudier le code source ou binaire d'un programme sans l'exécuter. L'analyse dynamique est l'opposé, c'est l'étude du code binaire à l'exécution, à travers différentes instrumentations. Ces méthodes ont pour but de détecter des erreurs de programmation ou de conception en étudiant le code. Une partie des erreurs détectées concerne les vulnérabilités sur l'utilisation mémoire, et plus précisément, les corruptions de mémoire. C'est dans ce cadre qu'il est intéressant de relever l'utilité de ces méthodes. En effet, même si l'analyse statique de code est capable de trouver des dépassements de tampon ou de mauvaises gestions de la mémoire plus rapidement que l'analyse dynamique, elle peut aussi aider à améliorer la qualité du code, la rapidité du code ou même forcer le respect d'un standard. À travers les trois prochaines sections, nous allons expliquer en quoi l'analyse statique, l'analyse dynamique et le fuzzing peuvent aider à combattre les corruptions de mémoire. L'idée n'est pas d'entrer dans le détail du fonctionnement de ces méthodes, car ce n'est pas sur ces solutions que nous allons apporter des contributions, mais plutôt de donner un rapide aperçu de ce qu'il est possible de faire, et montrer que c'est un complément indispensable aux autres protections.

Analyse statique

L'analyse statique de code couvre une grande variété de méthodes utilisées pour obtenir des informations sur le comportement d'un programme sans l'exécuter. De la même manière que pour la protection FORTITY_SOURCE en mode 1, l'analyse statique permet de détecter les vulnérabilités dans le code source. Cependant, les outils d'analyse statique actuels sont plus poussés que la protection de GCC. Ils vont être capables de détecter des vulnérabilités plus complexes qu'un dépassement de tampon sur une fonction *strcpy* par exemple. Il est difficile de faire une liste complète sur les possibilités de ces outils, car chacun possède ses avantages et ses inconvénients et il est parfois nécessaire d'en combiner plusieurs pour avoir une couverture optimale. De plus, certaines méthodes d'analyse statique utilisent le programme compilé au lieu du code source. Nous ne fournissons donc pas un état des lieux complet sur ces méthodes et outils d'analyse statique, mais l'état de l'art le couvre déjà [68]. Les travaux de K. Goseva *et al.* [69] sur la capacité des outils d'analyse statique à détecter des vulnérabilités concluent que les outils à l'état de l'art ne sont pas très efficaces. Dans leur expérimentation, 27% des vulnérabilités dans des programmes C/C++ du jeu de données Juliet [70] ne sont pas détectées par les outils. Ce fort taux d'erreur est dû à la complexité de couverture de code lors de l'analyse statique ou encore à la difficulté de détecter des vulnérabilités complexes sur le management du tas par exemple. Ce genre de techniques reste très intéressant pour éliminer un maximum de vulnérabilités du code, sans ajouter de surcharge sur ce dernier.

Analyse dynamique

À la différence de l'analyse statique, l'analyse dynamique étudie le comportement du programme compilé en l'exécutant. Cette méthode permet d'avoir un meilleur contexte lors de la vérification puisque l'exécution du programme va prendre en compte son environnement. L'analyse dynamique peut être utilisée pour profiler un programme, *i.e.*, avoir des informations sur le temps d'utilisation du processeur, l'utilisation de la mémoire ou encore l'énergie dépensée par le programme. L'analyse dynamique permet aussi de déboguer un programme en temps réel, en donnant la possibilité d'étudier la mémoire et l'état du processeur à n'importe quel moment de son exécution. Enfin, certains outils comme Valgrind [71] ou Address Sanitizer (ASAN) [72] permettent également de trouver des vulnérabilités dans les programmes, par exemple, l'accès à des zones mémoires interdites, ou des utilisations de mémoire non initialisées.

Fuzzing

Le fuzzing est une méthode d'analyse dynamique qui va tester le programme avec des jeux de données aléatoires. L'idée est de faire échouer l'exécution du programme en injectant des données aléatoires dans les points d'entrée pour générer un crash ou une erreur. Si ce cas arrive, cela veut dire qu'il y a des vulnérabilités à corriger. Le grand avantage du fuzzing est que la mise en place est extrêmement simple, ne demande aucune connaissance du fonctionnement du système et permet donc de trouver des vulnérabilités facilement et automatiquement. De plus, il existe des outils gratuits et très faciles d'utilisation comme American Fuzzy Lop (AFL) [73] proposé par Google ou Radamsa [74], qui permettent de mettre rapidement en place une politique de fuzzing intelligent pour tester un programme.

Certaines plateformes mettent en place toutes ces politiques pour créer des environnements de tests et d'attaques plus ou moins automatisés. C'est par exemple le cas d'angr [75] ou encore BAP [76] qui offrent tout un ensemble d'outils d'analyse statique, dynamique ou encore fuzzing. Ces outils d'analyse permettent donc aux développeurs de trouver et de supprimer certains bogues et vulnérabilités dans le programme avant sa mise en production.

3.2.2 Empêchement d'attaques

Dans cette sous-section, nous détaillons les protections qui sont déployées pour empêcher des attaques et qui agissent dans le cas où les méthodes précédentes laissent passer des vulnérabilités.

3.2.2.a Data Execution Prevention

La protection Data Execution Prevention (DEP) a été introduite au début des années 2000 dans les systèmes d'exploitation. Le premier patch à implémenter cette solution

de manière logicielle était PaX [77] qui ciblait le noyau Linux. La solution consiste à rendre non exécutable toutes les sections dans lesquelles il est possible d'écrire et vice versa. Le mode est géré par un bit appelé le NX bit. Plus tard, ces protections ont été déployées directement dans les processeurs et c'est le système d'exploitation qui eut la responsabilité de gérer ce bit. Par exemple, ARM implémente le bit NX [78] directement sur leur processeur. En 2004, Windows a aussi déployé le DEP [79] sur son système d'exploitation. En conséquence, cette solution, qu'elle soit logicielle ou matérielle, rend la pile et le tas non exécutables et les attaquants ne peuvent plus injecter de shellcode directement dans ces sections.

3.2.2.b RELocation Read Only

La protection *RELocation Read Only* (RELRO) [80] est une protection mise en place par GCC, permettant de demander au *linker* de résoudre les fonctions de bibliothèques dynamiques au tout début de l'exécution. L'idée est de pouvoir remapper la section Global Offset Table (GOT) en lecture seule pour éviter qu'un utilisateur malintentionné vienne écraser ces sections par des pointeurs malicieux. En effet, si la protection RELRO n'est pas activée, le calcul des pointeurs sur les fonctions des bibliothèques dynamiques n'est pas fait au début de l'exécution, mais pendant l'exécution. Il est donc possible d'écrire dans ces sections. De nombreuses attaques sur le tas, ou même les attaques par formatage de chaînes de caractères reposent sur l'écrasement d'une entrée dans la GOT. Il existe deux modes principaux pour cette solution:

1. RELRO partiel : les non-PLT (Procedure Linkage Table) passent en lecture seule, mais la GOT est toujours en écriture.
2. RELRO complet : support des caractéristiques du RELRO partiel et la GOT passe en lecture seule.

La solution 1 répond donc partiellement au problème, car les attaques sur la GOT restent possibles. La solution complète semble plus pertinente, mais ralentit fortement le lancement d'une application, car elle doit pré-lier toutes les fonctions des librairies dynamiques. Cependant, malgré l'effort fourni autour du RELRO, les tables de destruction DTORS (appelées juste avant exit) restent en écriture. De plus, pour accélérer le lancement des programmes utilisant une librairie partagée, ces dernières sont généralement compilées en RELRO partiel. Il en résulte que même si le programme est compilé en RELRO complet, mais que la librairie, par exemple la libc, est compilée en RELRO partiel, l'attaquant peut toujours attaquer la GOT de la libc en écrasant *__malloc_hook* ou *__free_hook*.

3.2.2.c Address Space Layout Randomization

L'Address Space Layout Randomization (ASLR) [81] est une technique permettant de placer de façon aléatoire les espaces d'adressage mémoire dans la mémoire virtuelle. Cette protection est déployée dans tous les systèmes d'exploitation modernes tels que

Windows, Linux, iOS, ou encore Android. L'idée est qu'à chaque chargement du binaire, la position de base du tas, de la pile et des bibliothèques change. Sur un système 32 bits, un bloc aléatoire de 12 à 16 bits est utilisé. Cette approche de base est représentée à la figure 3.4, où l'adresse de départ des librairies, et les segments de données sont déplacés entre les lancements consécutifs de l'application. De cette manière, un adversaire doit donc deviner l'emplacement des fonctions et des séquences d'instructions nécessaires au succès du déploiement de son attaque par réutilisation de code. Cette fonctionnalité rend l'injection de code beaucoup plus compliquée. Malheureusement, les implémentations actuelles de l'ASLR souffrent de deux principaux problèmes:

1. L'entropie des systèmes 32 bits est trop faible, et l'ASLR peut donc être contournée au moyen d'attaques par bruteforce [82, 83].
2. L'ASLR devient vulnérable si l'attaquant est capable de faire fuiter des données mémoire [84, 85] où l'adversaire prend connaissance d'une seule adresse d'exécution et utilise ces informations pour retrouver l'adresse de base d'une section. Cette technique est possible, car l'ASLR ne fait que randomiser l'adresse de base d'une section, la position des fonctions, codes, données ou variables à l'intérieur de ces sections reste identique.

Il en résulte que, quelle que soit la méthode utilisée, si l'attaquant est capable de retrouver l'adresse de base de la section, l'adversaire n'a plus qu'à ajuster dynamiquement tous les pointeurs à l'intérieur de la charge utile d'injection de code. Il existe donc plusieurs cas où cette solution peut être contournée.

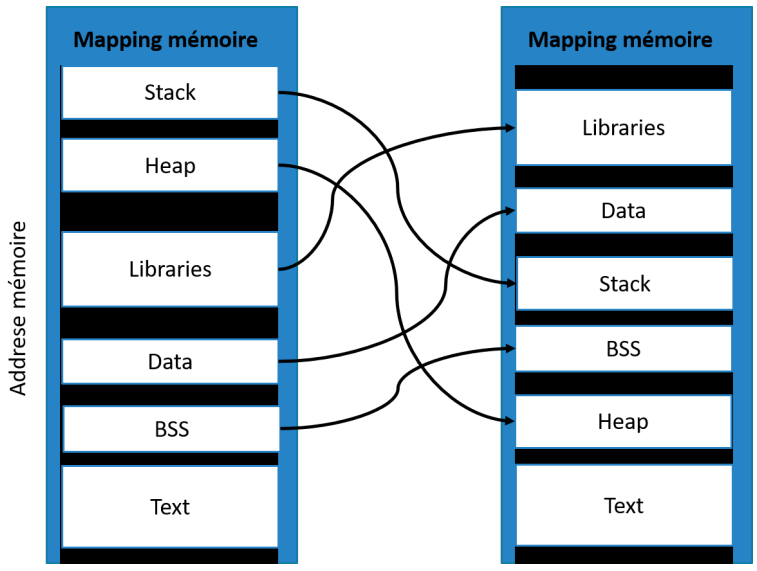

FIGURE 3.4 – *Adress Space Layout Randomization*

3.2.2.d Position Independent Executable

La protection Position Independent Executable (PIE) [86] est identique à l'ASLR, sauf que dans ce cas c'est la section TEXT du binaire qui est randomisée. Cette solution

complique encore un peu plus la tâche de l'attaquant lorsqu'il doit utiliser des gadgets pour faire fuiter l'adresse de base d'une section, car l'adresse des gadgets devient inconnue. Il est quand même toujours possible d'utiliser les attaques par bruteforce ou de faire fuiter des données autrement, en utilisant une attaque par formatage de chaîne de caractères par exemple. Cependant, la protection PIE reste un très bon complément de l'ASLR [87].

Au final, nous avons vu que les méthodes de prévention agissent aussi bien directement sur le code source du programme pour éliminer des vulnérabilités que sur le programme compilé lui-même pour empêcher certains types d'attaques. Néanmoins, comme nous l'avons vu, les techniques actuelles de prévention ne sont pas suffisantes pour contrer toutes les attaques de corruption de mémoire. L'analyse de code ne peut supprimer qu'une partie des vecteurs d'attaques du programme. Le DEP peut être contourné à l'aide d'attaques par réutilisation de code. En effet, comme la technique de réutilisation de code injecte des morceaux de code déjà exécutables contenus dans le binaire, il n'y a aucun besoin d'injecter un shellcode. Les techniques comme l'ASLR et le PIE sont facilement contournables à l'aide d'une fuite de données. Il en résulte qu'il est nécessaire d'aborder des méthodes alternatives.

3.3 Tolérance aux attaques

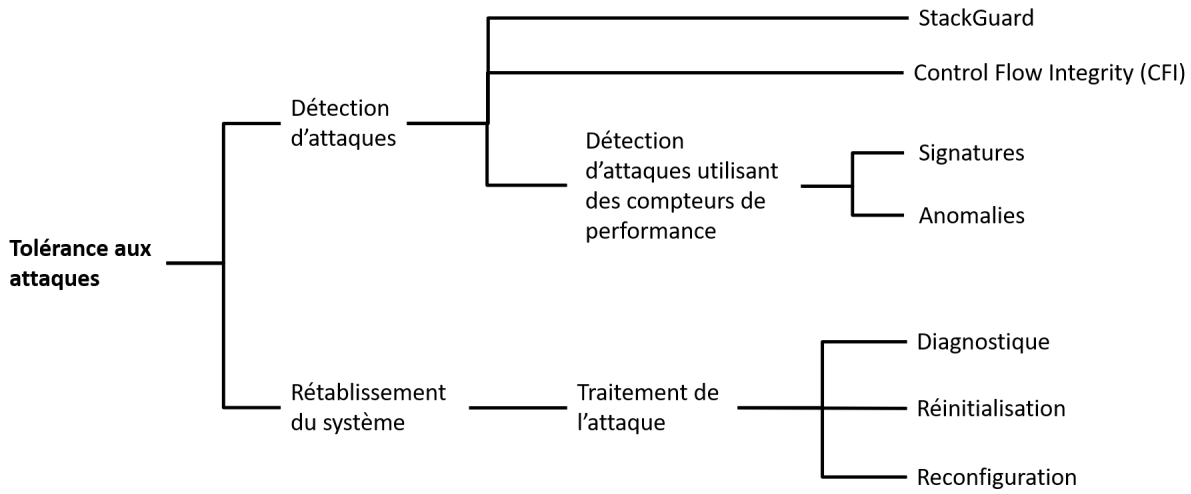

FIGURE 3.5 – *Classification des méthodes de tolérance aux attaques.*

Comme nous venons de le voir, les techniques de prévention actuelles ne sont pas suffisantes pour sécuriser un système face à toutes les attaques et vulnérabilités existantes. Il est alors nécessaire d'envisager un autre niveau de protection: la tolérance aux attaques. L'idée est d'agir une fois que la prévention a échoué, c'est-à-dire que le système subit une attaque. Celle-ci fonctionne en deux étapes principales qui sont (voir figure 3.5) :

- **Les méthodes de détection d'attaques** : elles sont des moyens d'identifier un comportement anormal dans un processus en cours d'exécution. Par exemple, un changement non autorisé dans le flot de contrôle du programme.

- **Le rétablissement du système** : Une fois l'attaque détectée, il est nécessaire de traiter l'information. Cela passe par des étapes de diagnostic qui visent à prendre une décision quant à la contremesure à appliquer. Par exemple, une action de réinitialisation ou une reconfiguration du système pour éviter l'attaque.

Dans cette section, nous nous concentrons sur la première étape qui concerne les méthodes de détection d'attaques.

3.3.1 StackGuard

La technique du Stackguard, proposé par Cowan *et al.* [88], implémenté dans GCC [89], a pour but de détecter des dépassements de tampon sur la pile. Elle consiste à insérer une valeur aléatoire avant la valeur de retour dans la pile. On appelle cette valeur un *canari* ou un *cookie*.

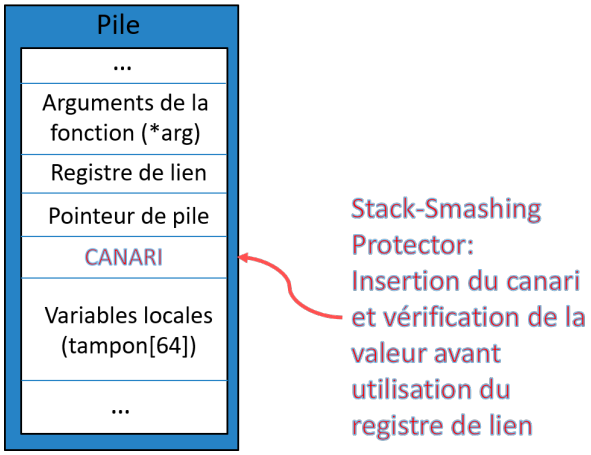

FIGURE 3.6 – *Stackguard*

La figure 3.6 illustre le fonctionnement de la protection. Lors d'un dépassement de tampon, la valeur du canari est écrasée lorsque l'attaquant essaie d'atteindre la sauvegarde de l'adresse de retour sur la pile. Avant d'utiliser cette sauvegarde, le système d'exploitation vérifie si le canari n'a pas changé. Si une modification est détectée, l'OS arrête le programme pour empêcher le succès de l'attaque. Un contournement possible est de trouver une méthode de fuite de données pour remplacer le canari par la valeur voulue lors du dépassement de tampon. Une fuite de données est généralement réalisée à l'aide de primitives de fuite d'adresse arbitraire. Il existe plusieurs méthodes pour faire fuiter des données. Il est par exemple possible d'écraser le dernier octet nul d'une chaîne de caractères pour faire fuiter les données présentes à la suite de la chaîne avec un *printf*. Une autre méthode peut être d'utiliser une erreur existante sur les fonctions de la famille *printf*. On appelle l'exploitation de ces vulnérabilités une attaque "format string" [90]. Le Stackguard est donc intéressant pour prévenir des dépassements de tampon sur la pile, cependant il contient des faiblesses qui peuvent le rendre inefficace.

3.3.2 Control Flow Integrity

La protection par Control Flow Integrity (CFI) est une technique de sécurité qui détecte les attaques par corruption de mémoire en surveillant le graphe du flot de contrôle (CFG) du programme. La méthode a été introduite pour la première fois en 2005 par Abadi *et al.* [91]. Ils ont proposé une approche basée sur des labels, où chaque noeud du CFG est marqué avec un label unique placé au début d'un bloc de base. À l'exécution, les labels sont comparés avec ceux du CFG. Depuis que l'idée initiale des CFI et le premier prototype ont été présentés en 2005, une pléthore de méthodes alternatives de type CFI a été proposée et mise en œuvre. Bien que toutes ces alternatives modifient légèrement le concept de base, elles tentent toutes de mettre en œuvre une politique d'intégrité de flot. Nous allons voir que cette politique peut être déclinée de plusieurs manières, que ce soit au niveau de la granularité des vérifications, de la nature des implémentations ou de la méthode de suivi du graphe.

3.3.2.a Principe

L'objectif de la mise en place d'un CFI est de restreindre l'ensemble des transferts possibles de flot de contrôle en suivant le graphe du binaire (CFG). Tout mécanisme de CFI comprend deux composantes: la composante d'analyse (souvent statique) qui récupère le graphe du flot de contrôle de l'application et la composante dynamique qui restreint les transferts entre blocs de base en fonction du CFG généré. La Figure 3.7 illustre le fonctionnement haut niveau général d'un CFI. Premièrement, et avant toute exécution, une étude statique du programme est faite pour construire son graphe de flot de contrôle (CFG). Ensuite, une vérification est faite à chaque instruction sortante ou entrante d'un bloc de base. Ces vérifications sont effectuées à l'exécution et visent à détecter un changement dans le flot de contrôle du programme. Par exemple, sur le CFG de l'image 3.7, un transfert normal est de passer du bloc de base BB_1 au bloc de base BB_3. Cependant, si le flot courant veut passer du BB_3 au BB_5 (une attaque ROP par exemple), comme ce transfert n'existe pas dans le CFG, le CFI détecte le changement durant l'exécution et alerte le processus en charge du rétablissement du système pour non respect du CFG.

Bien que dans la théorie cette méthode soit très prometteuse, dans la pratique il existe une pléthore de mises en oeuvre différentes, chacune avec ses avantages et ses inconvénients. Par exemple, les stratégies de CFI sur la composante dynamique sont divisées en deux catégories:

1. Une politique de vérification en avant qui gère toutes les instructions du type *call* et *jump*.

2. Une politique de vérification en arrière qui gère toutes les instructions du type *ret*.

En face de ces stratégies, il existe deux grandes familles d'implémentation:

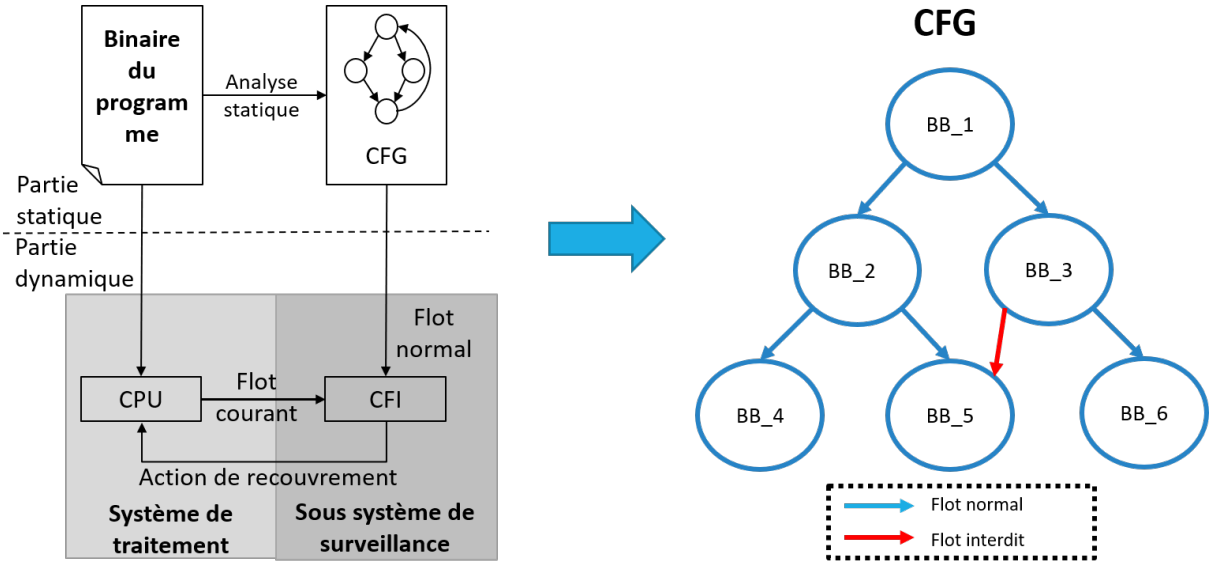

FIGURE 3.7 – *Control Flow Integrity*

1. Des implémentations logicielles, qui vont reposer sur des instrumentations de code au moment de la compilation par exemple.
2. Des implémentations matérielles qui vont se décliner elles-mêmes en deux versions:
   (a) Les méthodes à base de modification du pipeline du processeur pour récupérer les informations nécessaires à la mise en place du CFI.
   (b) Les méthodes qui vont utiliser les sondes de débog déjà présentes sur le processeur pour récupérer les informations nécessaires à la mise en place du CFI.

À partir de ces différentes stratégies et des choix d'implémentation, il s'impose de choisir la granularité des vérifications appliquée par le CFI. Les méthodes qui vont vérifier à l'échelle de l'instruction vont plutôt être classées dans l'ensemble des CFI dits à "grain fin" alors que celle dont les vérifications seront plus haut niveau (à l'échelle du bloc de base par exemple) seront dites à "grain épais". Ces modes ont des répercussions directes sur le compromis sécurité/coût de la sécurité.

3.3.2.b Implémentations logicielles

Les CFI logiciels s'appuient sur l'instrumentation de codes binaires au moment de la compilation. Cette méthode a l'avantage d'être facilement déployable sur n'importe quel système, car elle repose sur des fonctionnalités logicielles. Microsoft a récemment publié l'outil Control Flow Guard [92], déployé sur Windows. Il se concentre sur la protection des appels indirects, c'est-à-dire sur la stratégie de vérification en avant. L'approche consiste à calculer toutes les adresses cibles possibles lors d'un transfert indirect et à n'autoriser que ces transferts. Cependant, pour des raisons de performance, le Control Flow Guard ne met pas en œuvre une politique à grain fin, mais plutôt une

politique de vérification à grain épais. Il ne spécifie pas avec précision les adresses de saut valides pour chaque saut indirect. Au lieu de cela, toutes les adresses de saut valides sont incluses dans un ensemble global. De plus, le Control Flow Guard de Microsoft ne contrôle pas les sauts en arrière ; il reste donc vulnérable aux modifications du flot de contrôle avec des instructions *ret*. Le Control Flow Guard de Microsoft n'est pas la seule protection développée dans l'ensemble des CFI logiciels. Une multitude de projets ont vu le jour. V. Pappas a présenté kBouncer [93], un outil permettant d'atténuer les attaques de type Return Oriented Programming. ROPecker a été construit sur les idées de kBouncer [94]. Cependant, il vérifie plus souvent et plus minutieusement le flot d'exécution que kBouncer. LLVM-CFI [95] implémente une politique de vérification en avant alors que RAP [96] implémente les deux politiques. Le premier est une fonctionnalité proposée directement dans le compilateur CLANG alors que le second est un patch sur le kernel Linux. LLVM-CFI et RAP implémentent tous les deux un CFI basé sur des correspondances de prototype. Cependant, LLVM-CFI vérifie les adresses cibles autorisées alors que RAP autorise tous les transferts ce qui peut introduire une imprécision (de la même manière que le Control Flow Guard de Windows). D'un autre côté RAP impacte peu les performances, *i.e.* <1% alors que LLVM-CFI peut coûter jusqu'à 8.7%. J.Li *et al.* et X.Ge *et al.* ont tous deux présenté un CFI à grains fins pour la sécurité du noyau d'un système d'exploitation [97, 98]. De nombreux autres CFI logiciels ont été proposés au niveau académique, *e.g.*, Control Flow Locking [99], CCFIR [100], Bin-CFI [101], Strato [102], MIP [103], FECFI [104], CPI [105], PICFI [106], CSCFI [107], CCFI [108]. Ils se différencient par rapport à la granularité des vérifications, aux architectures ciblées, ou encore à leur manière d'implémenter les solutions.

L'idée n'est pas de venir faire une explication du fonctionnement de tous ces CFI un à un, mais de comprendre globalement quelles en sont les limitations. Comme nous l'avons vu, certaines implémentations sont à grain fin tandis que d'autres sont à grain épais. Plusieurs travaux ont montré les limitations des CFI à grain épais comme mécanisme de défense face aux attaques ROP. Par exemple, L. Davi *et al.* [109] fournissent une compréhension détaillée des solutions de CFI (kBouncer, ROPecker, ROP-Guard) et démontrent que ces implémentations restent vulnérables en introduisant plusieurs nouvelles primitives d'attaques ROP pour contourner ces CFI. Ces implémentations sont à grain épais et certaines, comme ROP-Guard, ne vérifient le flot de contrôle que lors de l'appel à certaines fonctions critiques. F. Schuster *et al.* [110] montrent aussi que ces CFI sont contournables et fournissent plusieurs démonstrations et preuves de concept sur des logiciels très utilisés tels que Internet Explorer ou Windows. Une autre étude, menée par E. Göktas *et al.* [111], a aussi démontré la possibilité d'attaques ROP sur les CFI logiciels. Les attaques sont démontrées sur Internet Explorer et Windows également, avec d'autres protections activées comme le DEP ou encore l'ASLR. Ils s'attaquent a l'un des CFI logiciels les plus restrictifs: CCFIR [100]. Ils expliquent comment abuser de ces restrictions pour construire un exploit en deux temps: un ROP pour créer une zone mémoire exécutable puis une injection de shellcode. Malgré leur succès à démontrer une attaque fonctionnelle sur un CFI très restrictif, ils concluent que la solution augmente encore la difficulté d'exploitation pour les attaquants. On note cependant que les implémentations à grain fin rencontrent des pénalités de performance allant de 8% à plus de 200%. Si l'on se base sur le constat que les implémentations logicielles sont soient contournables, soit trop lourdes, soit les deux, alors il est difficile de pouvoir faire un choix. Si l'on peut se permettre de déployer un CFI avec de faibles

pénalités pour une couverture partielle (comme pour la solution du StackGuard par exemple), il n'est pas envisageable de déployer des approches qui divisent par deux ou trois les performances du programme. C'est pourquoi les études se sont penchées sur les méthodes matérielles, pour adresser ce problème de performance tout en gardant un grain très fin.

3.3.2.c Implémentations matérielles

Dans l'approche, les CFI matériels sont similaires aux implémentations logicielles. La principale différence est qu'ils ne s'appuient pas forcément sur l'instrumentation de code. Les CFI matériels s'appuient sur la modification du matériel, en particulier la modification du pipeline du processeur pour extraire le flux d'information et suivre le flot d'exécution. Le flux d'information peut prendre différentes formes et dépend fortement de l'implémentation du module de vérification du CFI. Cela peut être des informations sur l'adresse de l'instruction exécutée, l'opcode de l'instruction, l'ajout d'un label, ou encore la valeur des registres du processeur. Certaines implémentations ne demandent pas de modification du pipeline, mais utilisent le débogueur en place pour extraire le flux d'information requis. Sur la base de ces approches, de nombreuses implémentations de CFI différentes peuvent être construites. De Clercq *et al.* présentent une analyse détaillée des politiques de sécurité appliquées par 21 architectures de CFI récentes basées sur des implémentations matérielles [112]. Par exemple, un Shadow Call Stack (SCS) est utilisé dans la plupart des solutions pour appliquer une politique de vérification en arrière [113, 114, 115, 116, 117], seule la politique de vérification en avant du flot de contrôle change. Le but d'un SCS est de détecter une adresse de retour altérée/écrasée qui est stockée sur la pile lors d'un appel de fonction, *i.e.*, un dépassement de tampon sur la pile. D'autre part, nous avons vu que pour avoir un CFI efficace, il faut combiner la politique de vérification en avant et celle de vérification en arrière. Si, de par son faible coût et ses pénalités quasiment nulles, le SCS est une solution unanime pour la politique de vérification en arrière, aucune méthode ne tire son épingle du jeu pour la politique de vérification en avant. Sullivan *et al.* ont mis en œuvre une approche basée sur les labels pour vérifier si le transfert entre blocs de base est correct [113]. Chaque bloc de base se voit attribuer un label, et le label est vérifié à chaque changement de bloc. Arora *et al.* ont proposé une architecture pour mettre en œuvre un CFI à grain fin protégeant le flot de contrôle intra-procédural en utilisant des tables de correspondance pour vérifier les transferts, tandis que le flot de contrôle inter-procédural est protégé en utilisant une machine à états finis [115]. Mao *et al.* ont proposé de dériver un flux d'information (composé de *opcode*, adresse, instructions *load/store*, flot de contrôle, hachage) du programme en cours d'exécution vers un second processeur pour détecter une attaque en comparant le flux d'information d'exécution avec celui calculé hors ligne [114]. Christoulakis *et al.* proposent HCFI, une implémentation de CFI sur un SoC SPARC qui repose sur la modification du pipeline [118].

Dans un autre genre, Wahab *et al.* proposent ARMHEx, une implémentation de Dynamic Information Flow Tracking (DIFT), qui surveille le flot de données et détectent un accès non autorisé en utilisant le débogueur en place sur le processeur et une implémentation sur FPGA [119]. CFIMon [120] exploite le Branch Trace Store (BTS) des

systèmes x86 pour obtenir les adresses sources et cibles des branchements. Ici, l'objectif est de vérifier les branchements effectués avec une base de données préconstruite de branchements légitimes afin de détecter les attaques sur le flot de contrôle.

Encore une fois, l'idée n'est pas de refaire une étude détaillée de toutes les implémentations des CFI matériels, De Clercq *et al.* fournissent déjà une analyse précise. Il est cependant nécessaire de comprendre les avantages et points faibles de ces solutions. Chacune des implémentations présente une faible surcharge et une protection efficace. Cependant, la modification du pipeline d'un processeur est un premier problème dans le déploiement matériel de CFI. Par exemple, les processeurs déjà déployés ne peuvent pas bénéficier de la solution et la solution ne permet donc pas de protéger les flottes de systèmes déjà déployées. Alors qu'une approche logicielle s'adresse à un plus large public et permet de protéger n'importe quel système. Donc, l'approche CFI matériel, bien qu'intéressante, ne convient pas au cahier des charges de ces travaux de recherche. Cependant, il est à noter que ARM et Intel travaillent tous deux à la mise en œuvre d'un CFI matériel directement intégré dans leur processeur. Intel propose la technologie Control-Flow Enforcement Technology (CET) [116] pour prévenir les attaques par réutilisation de code en utilisant des techniques de SCS et de suivi de branche indirecte. La technique repose sur une partie de modification matérielle, mais aussi sur l'instrumentation de code. ARM a défini une méthode pour protéger les pointeurs de code pour empêcher le détournement de flux de contrôle en protégeant l'intégrité des pointeurs de code (*Code Pointer Integrity* : CPI) [121] au moment de l'exécution. Cependant, le CET et le CPI sont encore en cours de spécification et les anciens processeurs ne bénéficieront pas de cette avancée. De nouvelles protections sont donc nécessaires.

3.3.2.d Conclusion

Les CFI logiciels et les CFI matériels permettent de détecter une attaque grâce à l'ajout de fonctionnalités spécifiques. Leur principal objectif est de surveiller le flot d'exécution du programme pour détecter une attaque. Alors que les CFI logiciels sont facilement extensibles, mais demandent un surcoût plus élevé, les implémentations matérielles impactent peu les performances, mais sont plus difficiles à mettre en place et sont évidemment plus coûteuses. Le déploiement des solutions doit être pris en considération lors du développement et les solutions logicielles permettent plus de flexibilité. De plus, les implémentations matérielles présentent deux inconvénients majeurs :

- Les processeurs déjà fabriqués ne bénéficieront pas de cette fonctionnalité.
- En cas de vulnérabilité de la solution matérielle, il n'est pas possible de la réparer.

En outre, les travaux sur "Control-Flow Bending" [122] mettent en évidence les limites des CFI en tant que mécanisme de protection en soi. Ils proposent la mise en place d'attaques sur des CFI à granularité très fine (donc protection maximale). L'idée est d'attaquer les données utilisées et non le flot de contrôle directement. Ainsi, une fonction *exec* (fonction Linux permettant d'exécuter des processus à partir d'un code C par exemple) appelée avec des arguments altérés ne va pas changer le flot d'exécution,

mais peut fortement changer le comportement du programme. En effet, comme un CFI ne suit sur le flot des blocs de base, *i.e.* les différents branchements du programme, ils sont incapables de détecter un changement dans les données. Pour sécuriser aussi bien le flot de contrôle que les données utilisées, une première alternative est d'utiliser des protections de type *Data Information Flow Tracking* (DIFT) comme proposé par Wahab *et al.*. Cependant, ces méthodes impactent encore plus les performances. Ainsi, il faut explorer d'autres mécanismes de détection.

3.3.3 Détection à l'aide de compteurs de performance

Les processeurs et microprocesseurs modernes possèdent des unités de surveillance de performance (*Performance Monitoring Unit* : PMU) qui ont pour but d'aider à profiler et optimiser l'utilisation des processeurs. Mais, leur utilisation peut aussi permettre de vérifier l'intégrité du flot de contrôle de manière indirecte. La solution est intéressante, car d'une part, elle repose sur des dispositifs matériels présents dans la plupart des processeurs et d'autre part, il y a besoin d'une partie logicielle pour initialiser et récupérer les informations produites par ces éléments matériels pour implémenter la vérification finale. La figure 3.8 illustre le fonctionnement global. Un modèle du fonctionnement du programme est construit hors ligne, puis embarqué sur le système final. Ce modèle peut aussi bien prendre la forme d'un seuil basé sur des valeurs à ne pas dépasser à un instant t de l'exécution que la forme d'un modèle de machine learning. Il est ensuite utilisé pour effectuer la vérification et déterminer si le système s'exécute correctement lors de son fonctionnement. La solution est donc différente des CFI matériels, car cette fois, il n'est pas nécessaire d'ajouter du matériel supplémentaire, mais aussi différente des CFI logiciels, car elle profite de l'accélération matérielle. Nous proposons d'en étudier les spécificités.

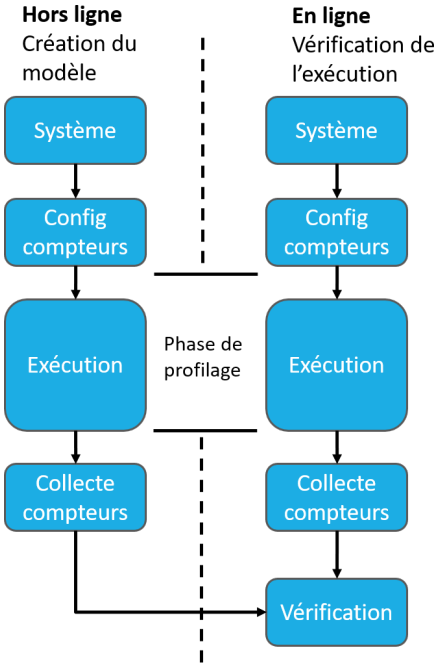

FIGURE 3.8 – *Control Flow Integrity avec l'utilisation de compteurs de performance.*

Alors que les méthodes de détection vues jusqu'à présent se concentrent sur l'étape d'exploitation de la vulnérabilité, celles-ci peuvent aussi détecter les attaques une fois que le logiciel malveillant est déjà en cours d'exécution, c'est-à-dire lors de l'étape post-exploitation. Ce type de méthode s'appuie sur des métriques indirectes pour déduire une potentielle attaque. L'avantage principal est qu'à l'instar des CFI qui ne détectent pas une attaque sur les données, les méthodes indirectes en sont capables. Nous proposons de les classer en deux catégories :

1. **La détection de signature :** cela signifie détecter une menace connue et qui est identifiable par rapport a son comportement. Les indicateurs utiles sont nombreux, par exemple, le *hash* d'un logiciel ou d'un fichier, l'IP d'une machine, ou encore une séquence d'octets.
2. **La détection d'anomalies :** on cherche à détecter une menace en se basant sur la connaissance du comportement normal du programme. Si ce comportement dévie de la normalité, alors une alerte est levée.

Nous proposons d'approfondir ces deux catégories pour mieux comprendre leurs principes et les différentes implémentations existantes. Nous proposons également de comparer et évaluer les méthodes existantes selon sept critères importants :

1. Le mode de détection, *i.e.* signature ou anomalie.
2. Le nombre de compteurs utilisés.
3. Le type de modèle utilisé.
4. Les pénalités en termes de performance.
5. Les pénalités en termes de stockage mémoire.
6. La précision de détection.
7. Le nombre d'échantillons utilisés et le taux d'échantillonnage.

3.3.3.a Principe de détection de signature

La détection de signatures peut s'apparenter à de la détection de malware. Dans la figure 3.9, nous expliquons le fonctionnement de la détection de signatures utilisant des HPC et des modèles basés sur le machine learning. Pour rappel, une attaque peut se détecter en deux points différents : lors de l'exploitation ou lors de l'exécution du logiciel malveillant. Dans le cas de la détection de signature, c'est généralement le comportement d'un logiciel malveillant qui est recherché, donc l'étape post exploitation. La protection se passe alors en deux phases. Lors d'une première phase, hors ligne, on construit le modèle de machine learning à l'aide des malwares connus (ou d'informations connues pour être utilisées par les malwares), des HPC, et d'une étape d'apprentissage. La deuxième phase utilise ce modèle et les données produites

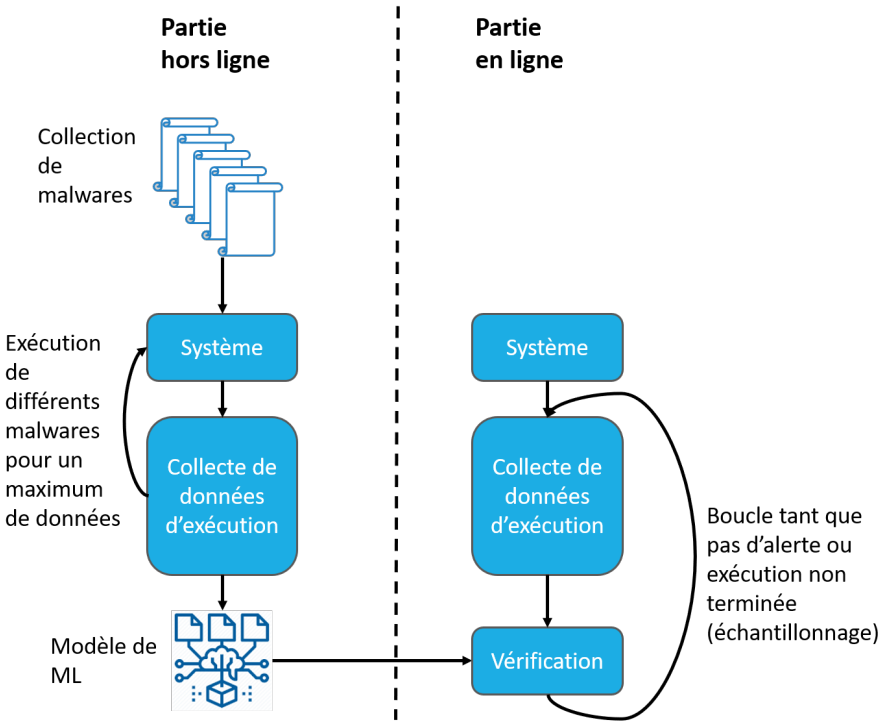

FIGURE 3.9 – *Détection de signature.*

à l'exécution du programme protégé pour trouver une corrélation avec les malwares connus. L'avantage est que le modèle créé fonctionne pour protéger n'importe quel programme, et qu'il n'est pas nécessaire de faire un apprentissage personnalisé au programme à protéger. Cependant, la méthode présente plusieurs inconvénients. Le premier est que, comme on recherche des signatures connues, il est peu probable de détecter des malwares inconnus. Le second est que la détection se fait forcément après l'étape d'exploitation, lorsque le malware commence à s'exécuter et a donc pu commencer à corrompre sérieusement la cible.

3.3.3.b Implémentations de détection de signature

Encore, une fois, plusieurs travaux ont implémenté ces concepts.

Demme *et al.* [123] tentent d'identifier l'exécution de malware après l'étape d'exploitation. Ils profilent l'exécution du programme en l'exécutant plusieurs fois, avec deux taux d'échantillonnage différents. Un à grain fin avec 25 000 cycles par échantillon et un à grain épais avec 50 000 cycles par échantillon. Ils ont expérimenté trois modèles différents de machine learning : KNN, arbres de décision (DT) et forêt aléatoire (RF). Ils n'ont pas donné de bons résultats en matière de précision dans la détection des logiciels malveillants dans la plupart des cas. Certaines familles de malware, comme *Tapsnake* ou *Zitmo*, sont cependant très bien détectée avec des taux de 100%. Aucune information n'est donnée sur les pénalités de la méthode.

Wang *et al.* [124] ont déporté le processus de machine learning sur un serveur pour traiter les échantillons des HPC et réduire la surcharge sur le système embarqué. Ils ont également compressé les données envoyées pour diminuer l'occupation de la bande

passante. Ils ont expérimenté leur outil sur un processeur AMD avec des rootkits Linux en utilisant 25 000 échantillons à grain fin (taux d'échantillonnage de 100 000 instructions) et à grain épais (taux d'échantillonnage de 300 000 instructions). Ils ont rapporté une surcharge de performance de 4,42%.

Zhou *et al.* [125] ont construit un ensemble de données contenant divers logiciels malveillants et bénins et ont utilisé les HPC pour tenter de les différencier. Ils ont testé différentes combinaisons d'évènement sur six compteurs, et de multiples modèles de machine learning: RF, KNN, NN, DT, AdaBoost et Naives Bayes. Aucune information n'est donnée sur les pénalités de leur méthode ou encore l'échantillonnage utilisé. La précision ne dépasse pas les 80.78%. Ils en concluent qu'il n'y a pas de lien de causalité entre les évènements micro-architecturaux de bas niveau et le comportement des logiciels de haut niveau pour faire de la détection de signature sur des malwares. Un comportement jugé mauvais dans certaines conditions pourrait être bon dans d'autres.

3.3.3.c Analyses des méthodes de détection de signature

Tableau 3.1 – *Récapitulatif des méthodes de détection de signatures (/ = Non spécifié).*

Travaux	**Mode de détection**	**Nombre de compteurs utilisés**	**Modèles utilisés**	**Pénalités de performance**	**Stockage mémoire requis**	**Précision de détection**	**Fréquence échantillonnage**
Demme (2013) [123]	Signature	/	DT, KNN, RF	/	/	Entre 25% et 100%	Toutes les 25k à 50k instructions
Wang (2016) [124]	Signature	/	1-SVM	4.42%	/	/	Toutes les 100k à 300k instructions
Zhou (2018) [125]	Signature	6	DT, KNN, RF, NN, AdaBoost, NB	/	/	80.78%	/

Le tableau 3.1 récapitule les critères des méthodes analysées et affiche des résultats mitigés. La méthode semble effectivement limitée dans l'approche même de la théorie. En effet, trop de contraintes sont présentes :

1. Il faut avoir une grande connaissance des malwares existants et créer un jeu de données en contenant le plus possible.

2. Il est très improbable de détecter une attaque non connue. Rien ne certifie que l'attaque fait partie du jeu d'apprentissage.

3. Même si les précisions annoncées sur la détection de certaines familles de malwares/d'attaques sont correctes (>98%), la moyenne sur toutes les familles n'est clairement pas intéressante (<80%).

Même si l'idée de déporter le traitement machine learning sur un serveur pour réduire drastiquement les surcoûts de performance et de stockage sur le système est intéressante, nous ne pensons pas que la technique par détection de signature soit la bonne approche. De plus, des comportements jugés comme inopportuns par certaines applications pourraient être corrects pour d'autres, ce qui ajoute un biais conséquent lors de l'apprentissage.

3.3.3.d Principe de détection d'anomalies

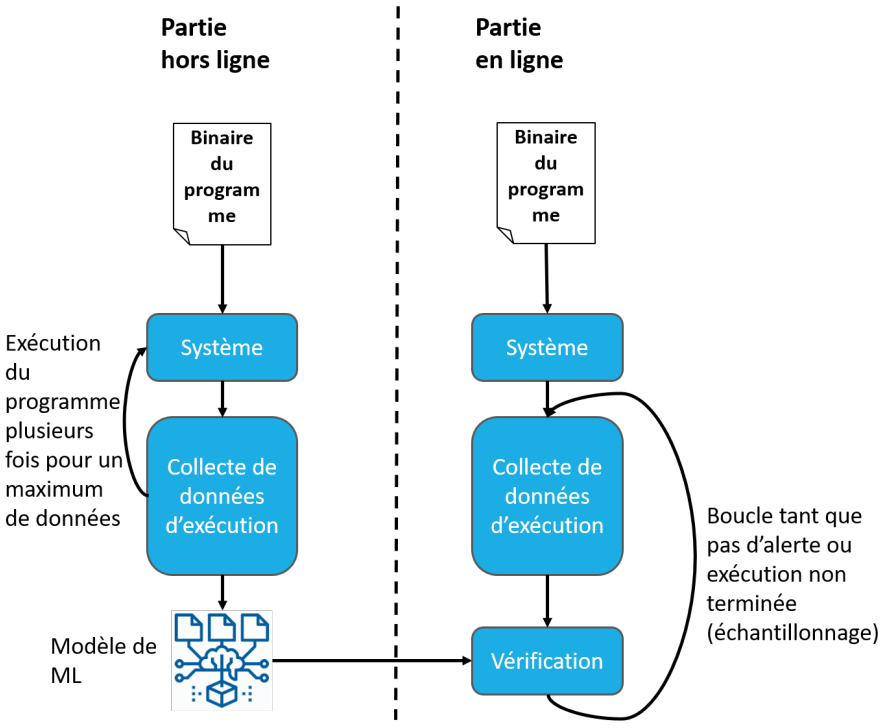

FIGURE 3.10 – *Détection d'anomalies.*

Le principe est très similaire aux CFI déjà présentés. Au lieu d'avoir une modification du pipeline du processeur ou une instrumentation du code pour pouvoir suivre le flot de l'exécution, les PMU, et plus particulièrement les compteurs de performance matériels (HPC), sont utilisés pour suivre indirectement le flot de contrôle. La figure 3.10 illustre le fonctionnement. Une partie hors ligne reste nécessaire et consiste à construire un modèle du programme basé sur les informations indirectes utilisées (dépendantes de l'implémentation). Ce modèle représente le comportement normal du programme. Le programme s'exécute ensuite sur le CPU, et impacte les données produites par les éléments matériels. Le modèle précédemment créé est alors utilisé pour

déterminer si le programme est dans un état normal à partir des données de l'exécution. Si un comportement anormal (une attaque ou un bug) est détecté, alors une alerte est levée. La méthode est finalement similaire à la détection de signatures. Seule la construction du modèle change : dans un cas il est construit sur la base du fonctionnement normal du programme et dans l'autre sur la base du fonctionnement des malwares. L'avantage de la méthode est qu'elle peut aussi bien détecter l'étape d'exploitation que l'étape post exploitation. D'autre part, la construction du modèle reste simple, car il repose seulement sur la connaissance du comportement normal du programme. De plus, cette approche rejoint finalement l'idée des CFI, qui peuvent détecter une attaque à partir de la comparaison du CFG et du flot d'exécution, avec l'avantage supplémentaire de détecter des attaques sur les données.

3.3.3.e Implémentations de détection d'anomalies

Plusieurs travaux sont partis de ce concept pour mettre en place la technique.

DCFI-checker [126] utilise des compteurs de performance pour compter le nombre d'instructions de branchement. Dans cette étude, la méthode surveille les instructions *call* et *ret* en insérant une sonde à la compilation (méthode intrusive) pour lire le nombre d'instructions de branchement exécutées. Les valeurs prévues par le modèle sont ensuite comparées avec les valeurs relevées pour détecter un changement dans le flot de contrôle. La solution a été testée sur le noyau Linux 2.6.15-1 et a fait état d'un surcoût de performance en fonction du nombre de sondes insérées, c'est-à-dire grain épais contre grain fin (respectivement 2,1% et 73,4%). Cette pénalité élevée en mode grain fin rend la solution non viable pour le déploiement.

ConFirm [127] est une technique intrusive (instrumentation logicielle) peu coûteuse, intégrée dans le code de démarrage d'un système embarqué, qui détecte des modifications du programme d'un système embarqué en utilisant des compteurs de performance matériels (évènements de branchements, de *load/store*, d'instructions exécutées). La détection se fait en deux étapes. Le premier profilage est effectué hors ligne en insérant des points de contrôle à des positions prédéterminées. Ensuite, à l'exécution, le profilage hors ligne est comparé avec le profilage en ligne, à l'aide des valeurs fournies par les HPC. Si les valeurs fournies dépassent un seuil à chaque étape du profilage, alors une modification malicieuse est diagnostiquée. Ils ont mis en œuvre la solution sur des architectures ARM et Power-PC. Ils ont rapporté des pénalités de performance de 8,48% sur la plateforme ARM Cortex-A15 et de 5,62% sur la plateforme PowerPC e300c3. La méthode requiert un stockage équivalent à 1% de la taille du binaire.

Malone *et al.* [128] utilisent des HPC pour détecter une attaque par corruption de mémoire. Ils ont défini deux types de profilage : un basé sur l'exécution complète du programme et l'autre basé sur un échantillonnage. Le profilage complet vérifie les compteurs à la fin du programme tandis que le profilage au moment de l'exécution vérifie les compteurs à un taux d'échantillonnage prédéterminé. Ils ont utilisé un modèle linéaire pour déterminer si le système fonctionne en toute sécurité ou s'il est attaqué. Ils l'ont testé sur un processeur Intel avec un ensemble de données d'environ

65 000 échantillons, représenté par des valeurs des compteurs lors de l'exécution des programmes protégés. Ils ont rapporté une surcharge de performance d'environ 10%, alors que leur surcharge de stockage est très faible (54 octets) en raison des modèles linéaires. Aucune précision n'est fournie, mais la méthode arrive à détecter l'attaque sur des programmes bien connus tels que *bzip2* ou *dijkstra*.

Tang *et al.* [129] ont utilisé les HPC pour détecter les logiciels malveillants en se basant sur des modèles de détection d'anomalies. Ils ont expérimenté leurs outils sur une plateforme Intel en utilisant le modèle de machine learning Support Vector Machine à classe unique (1-SVM). Ils ont profilé l'exécution avec un échantillonnage d'instructions à diverses granularités, de 16k instructions à 512k instructions. Ils ont également introduit la temporalité entre les échantillons. De cette façon, chaque échantillon dépend de N autres échantillons. Ils ont réussi à obtenir une très grande précision dans la détection du stade d'exploitation, mais avec un surcoût en performances élevé (de 40% à 100%). Cependant, la précision de la détection au stade de l'exécution du malware est très faible, inférieure à 50%.

3.3.3.f Analyses des méthodes de détection d'anomalies

Tableau 3.2 – *Récapitulatif des méthodes de détection d'anomalies (/ = Non spécifié).*

Travaux	**Mode de détection**	**Nombre de compteurs utilisés**	**Modèles utilisés**	**Pénalités de performance**	**Stockage mémoire requis**	**Précision de détection**	**Fréquence échantillonnage**
Malone (2011) [128]	Anomalie	2	Linéaire	10%+	54 octets	/	/
Tang (2014) [129]	Anomalie	2	1-SVM	40% à 100%	/	Entre 50% et 99%	Toutes les 16k à 512k instructions
DCFI-Checker (2014) [126]	Anomalie	2	Seuil	2.1% à 73.4%	/	/	/
ConFirm (2015) [127]	Anomalie	4	Seuil	5.62% à 8.48%	1% de la taille du binaire	/	Inséré aléatoirement

Le tableau 3.2 récapitule les critères des méthodes de détection d'anomalies analysées et les résultats semblent plus encourageants. Cependant les travaux actuels souffrent de limitations. Les travaux comme ConFirm ou DCFI-Checker reposent sur l'utilisation de seuils pour détecter une déviation de la valeur des compteurs. Cependant, le bruit inhérent au système, composé des différentes applications qui s'exécutent, de l'activité réseau, de la charge du processeur à un instant t, peut impacter les

compteurs. Ainsi, dans leur conclusion, les auteurs de ConFirm préconisent de s'orienter vers des algorithmes de machine learning pour traiter ces données et augmenter la précision de l'outil, tout en portant une attention particulière aux performances requises.

Cependant, les travaux utilisant les modèles de ML souffrent aussi de limitations. Ils n'ont ni poussé d'analyse sur le choix des évènements pour les compteurs ni sur le choix des modèles de machine learning. Une analyse plus poussée pourrait permettre d'améliorer la précision de détection et de baisser les pénalités de performance. De plus, d'autres pistes de recherches semblent intéressantes. Par exemple, séparer les coûts liés au profilage des compteurs des coûts liés aux modèles de machine learning. Ainsi il serait potentiellement possible de baisser drastiquement les performances des méthodes présentées en déportant le traitement des algorithmes de ML par exemple. L'échantillonnage est aussi très important, que ce soit le nombre d'échantillons nécessaire, ou la fréquence d'échantillonnage, une analyse plus poussée sur l'impact de ces réglages pourrait permettre de jouer sur le ratio pénalité/précision. Ces travaux fournissent donc une première intuition positive quant à la possibilité de détecter des attaques par corruption de mémoire à l'aide de HPC et de machine learning.

3.4 Synthèse

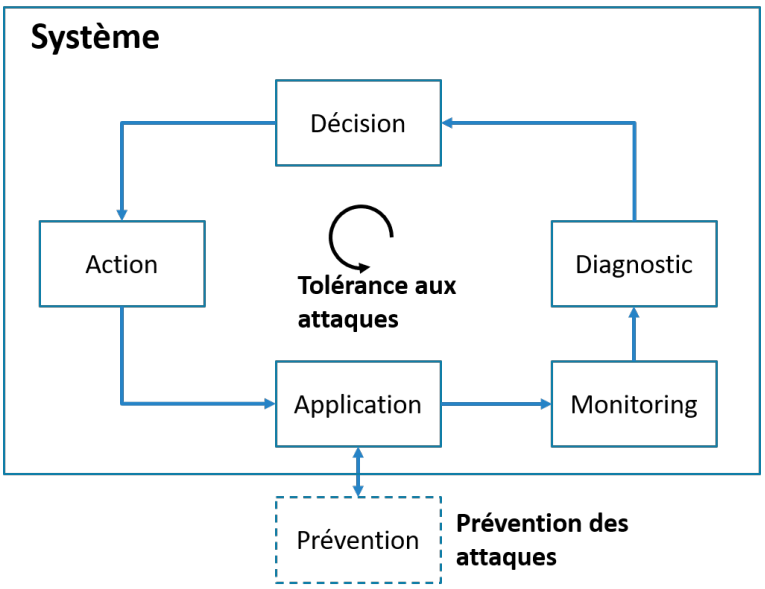

FIGURE 3.11 – *Boucle de sécurisation d'un système.*

Au cours de ce chapitre, nous avons proposé une classification des méthodes de sécurité visant à répondre à quatre axes nécessaires pour déployer un système sécurisé. Nous avons défini deux familles de méthodes : la prévention des attaques et la tolérance aux attaques. Au final, le développement de la sécurité d'un système peut être vu comme une boucle qui suit un cycle (figure 3.11). Dans un premier temps, l'idée est de mettre en place des techniques de prévention sur le système pour éliminer un maximum de vulnérabilités et empêcher certaines attaques, et ainsi créer une première ligne de défense. Plusieurs possibilités existent comme les protections des compilateurs, les

protections des systèmes d'exploitation, ou encore l'analyse statique, dynamique et le fuzzing. Bien que cette étape soit indispensable, elle ne permet pas de prévenir toutes les attaques existantes. C'est pourquoi il est aussi nécessaire que le système soit tolérant aux attaques et soit capable de les détecter pendant l'exécution. Nous avons vu différentes solutions de détection :

1. Le StackGuard qui permet de détecter un dépassement de tampon sur la pile.
2. Les CFI qui ont pour but de vérifier le flot d'exécution du programme. Ils se déclinent en version logicielle et en version matérielle.
3. Les techniques à base de compteurs de performance et de modèles pour détecter la signature d'un malware ou une anomalie dans le programme/système.

Toutes les techniques actuelles présentent des résultats intéressants, mais comme nous l'avons vu, souffrent de limitations. Le StackGuard peut être contourné si l'attaquant peut fuiter le canari et n'a qu'une politique de vérification en arrière. Les CFI logiciels demandent un surcoût beaucoup plus élevé en performances que les implémentations matérielles, mais ces dernières sont difficilement déployables. De plus ces méthodes ne sont pas capables de détecter des changements malicieux sur les données. Les méthodes basées sur la détection de signature n'ont pas donné de résultats concluants, et l'approche de la méthode n'est pas satisfaisante, *i.e.* connaître à l'avance le comportement de l'attaque. Cependant, les méthodes basées sur la détection d'anomalies présentent des résultats à creuser. Le choix des évènements, des modèles de machine learning, ou encore l'étude des performances face à la précision de détection sont des pistes de réflexion intéressantes, qui pourraient permettre d'améliorer ces techniques pour rentrer dans notre cahier des charges. L'étage de surveillance est donc un point crucial sur lequel se focaliser et nous proposons une contribution dans le chapitre suivant.

IV

Détection d'anomalies à l'exécution

Sommaire

4.1 Motivations

Dans le chapitre 3, nous avons proposé une classification des méthodes de sécurité et détaillé les méthodes de prévention et de détection d'attaques. Bien que les méthodes de prévention soient très intéressantes, ces mécanismes peuvent être contournés et nous ne pouvons donc pas leur faire entièrement confiance. Nous avons alors étudié les différentes méthodes de détection d'attaques, qui nous ont amené sur les solutions de détection d'anomalies à base de HPC. Cependant, les solutions actuelles, référencées dans le tableau 3.2, présentent des inconvénients non négligeables. Les solutions qui reposent uniquement sur un modèle à base de seuil pour procéder à la vérification des compteurs ne sont pas assez résistantes face au bruit d'un système. Les solutions basées sur le ML de Malone [128] et Tang [129] ont un coût en performance trop élevé, respectivement 10% et plus de 40%, et ne poussent pas assez loin l'analyse sur le choix des évènements et des modèles de ML à utiliser.

C'est dans ce cadre que nous proposons d'étudier plus en profondeur les méthodes de détection d'anomalies qui font partie de l'axe tolérance aux attaques (en vert sur la figure 4.1).

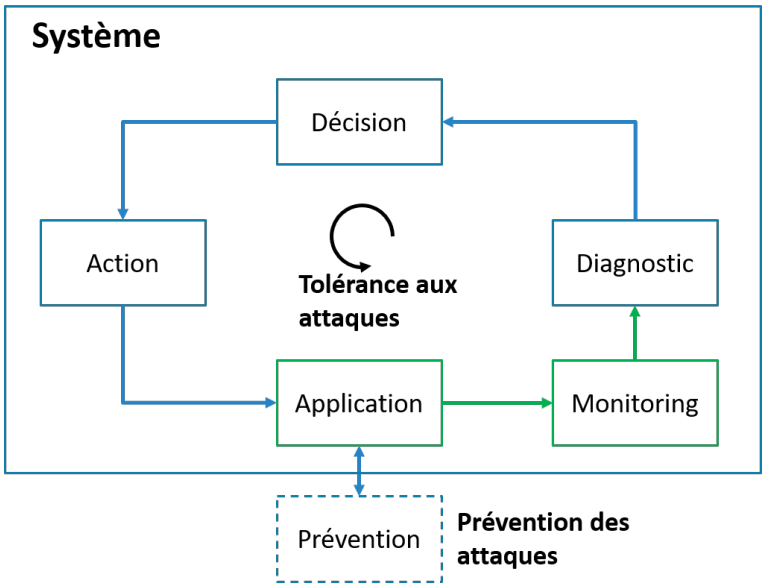

FIGURE 4.1 – *Boucle de sécurisation d'un système - Étage de monitoring ciblé.*

L'objectif principal est d'identifier un comportement anormal en utilisant les HPC et le ML, basé sur l'apprentissage de son comportement dit "normal", pour détecter une attaque par corruption de mémoire. Nous proposons d'analyser le choix des évènements et le choix des modèles de ML pour augmenter le ratio précision/pénalité, tout en surveillant le taux de faux positifs et de faux négatifs. Nous proposons également d'étudier une approche client/serveur de la solution pour minimiser les pénalités apportées par le machine learning sur le système. La technique est évaluée sous différents scénarios, pour en valider le fonctionnement.

4.1.1 Critères d'évaluation

Nos cibles prioritaires sont les systèmes embarqués dans l'IoT, c'est-à-dire que la solution doit fonctionner sur des microcontrôleurs et microprocesseurs. Pour avoir une solution la plus efficace et adaptative possible, nous posons plusieurs critères d'évaluation:

1. La solution doit être légère en matière d'exécution. Idéalement, ne pas dépasser 3% de pénalités sur les performances d'exécution. La mémoire utile au fonctionnement est aussi à étudier, puisque dans le cadre d'un système embarqué, cette ressource est généralement restreinte. De ce fait, elle est limitée.

2. La solution doit principalement fonctionner de manière logicielle, mais doit pouvoir être adaptée sur une version matérielle. On mettra ici en concurrence le coût financier de la solution face à la facilité de déploiement.

3. Toutes les fonctionnalités sur lesquelles reposera la solution devront être standards et disponibles aussi bien dans des microprocesseurs que dans des processeurs. Nous ne considérons pas l'ajout de fonctionnalités matérielles.

4. Le ratio précision/pénalité doit être réglable. C'est-à-dire qu'il serait possible de sacrifier de l'efficacité sur la détection pour gagner significativement en matière de performance.

5. La solution doit être orientée de manière à détecter une anomalie dans le comportement dit "normal" du système. Cette particularité permet à la solution de détecter des attaques inconnues, que ce soit sur le flot de contrôle ou les données, et donc d'élargir son champ d'action.

De plus, il est aussi évident qu'il faut évaluer la solution selon sa précision de détection. Cependant, ce paramètre est complexe à évaluer, car il existe une infinité de programmes et de systèmes à protéger et les attaques sont inconnues. Il devient alors complexe de donner des chiffres de précision de détection globaux, car la méthode repose sur l'apprentissage du comportement normal d'un système. Il faut donc ajouter cette problématique pour l'évaluation du problème.

4.2 Méthode de détection d'anomalies lors de l'exécution

4.2.1 Méthodologie

Le défi technique pour proposer un mécanisme de détection d'anomalies réside dans l'analyse précise du comportement du système, et/ou du programme, pour classifier l'exécution entre une activité normale et anormale. La méthode, basée sur l'utilisation de HPC et de modèle de ML, repose sur deux caractéristiques principales :

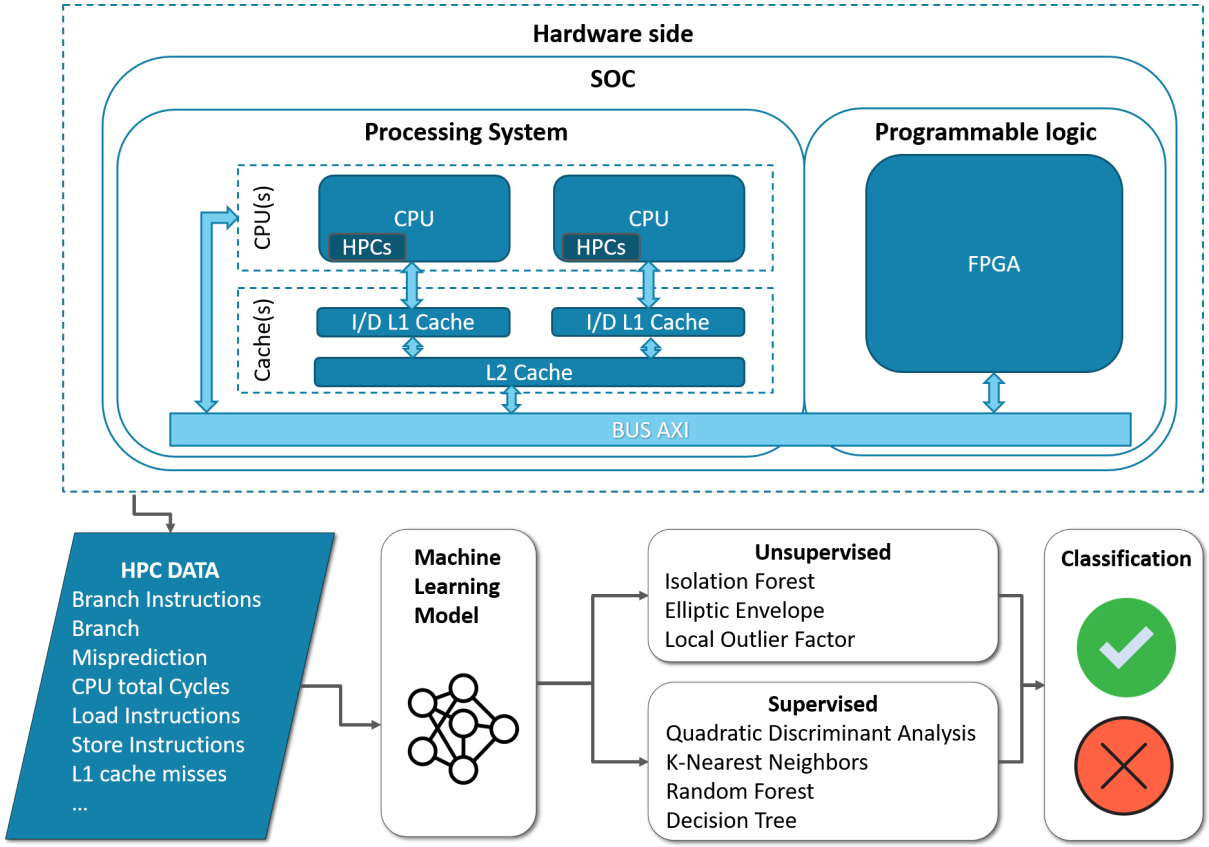

FIGURE 4.2 – *Méthodologie de la méthode de détection.*

1. **La sélection d'évènements** appropriés pour les compteurs matériels qui révèlent le maximum d'informations sur le comportement du programme.

2. **La sélection de modèles de machine learning** rapides qui peuvent effectuer une classification (attaque/pas attaque) avec une précision haute (au moins >98%), tout en offrant une faible surcharge de performance et un faible coût de stockage.

La méthode est illustrée par la figure 4.2. Le système recueille des informations comportementales sur l'exécution des processus à l'aide des HPC du processeur. Ces informations sont ensuite classifiées à l'aide de modèles de ML pour déterminer si le système est dans un comportement normal ou anormal. La technique comporte donc trois parties distinctes :

1. **Le profilage à l'exécution**, à l'aide des compteurs et des évènements.

2. **L'entraînement des modèles de ML**.

3. **La classification** pour détecter, ou non, une anomalie.

L'objectif de notre étude est mettre en place le mécanisme de détection et de l'évaluer dans des scénarios réalistes, sur du matériel de base, pour trouver le meilleur ratio performance/pénalité. Dans un premier temps, nous proposons de détailler le rôle et les problématiques de chacune de ces parties.

4.2.1.a Profilage à l'exécution

Au cours de cette première phase, les HPC enregistrent les valeurs des évènements sélectionnés pendant l'exécution. Deux approches sont possibles ici: suivre l'exécution du programme ou celle du système complet. Dans cette étude, nous avons sélectionné le premier cas pour diverses raisons :

1. Comme nous l'avons déjà expliqué dans le chapitre 3, Zhou *et al.* qui ont essayé de caractériser le système complet (PC utilisateurs) pour détecter un malware ont conclu que ce n'était pas possible à partir de seulement des valeurs de HPC. Il y a trop d'évènements dans le système qui impactent les compteurs et des comportements qui sont jugés corrects pour certains types d'applications sont jugés comme mauvais par d'autres.
2. Nous nous plaçons dans un contexte système embarqué, qui repose souvent sur l'exécution d'un service précis, donc suivre un seul programme semble être une bonne approche. Ici, le profilage d'un seul programme permet d'avoir une caractérisation optimale de ce dernier.
3. Une fois la technique validée sur l'approche programme, il reste possible de passer sur une approche système, mais il faut porter une attention particulière au nombre d'applications s'exécutant sur le système.

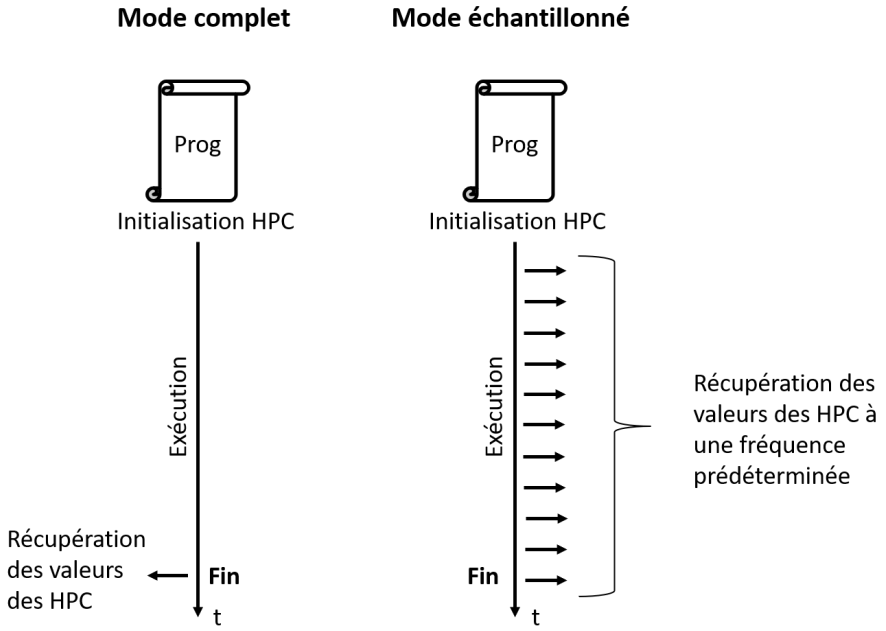

FIGURE 4.3 – *Profilage à l'exécution.*

Lors de l'exécution, les compteurs sont initialisés pour suivre le fonctionnement du programme à protéger et récupérer les valeurs des évènements sélectionnés. Le choix des évènements les plus pertinents pour caractériser le comportement normal est une question abordée dans la section 4.3.2.a, ici nous expliquons le fonctionnement global du profilage (figure 4.3). Le profilage est possible à travers deux modes, à savoir le mode *complet* et le mode *échantillonné*.

En mode *complet*, nous récupérons les informations comportementales d'un processus en accumulant les valeurs des compteurs sélectionnés à la fin de l'exécution du programme. Il présente l'avantage d'avoir très peu de pénalités de performance, car il ne récupère les valeurs des HPC qu'une seule fois durant l'exécution du programme. Cependant, le mécanisme de détection peut avoir plus d'avantages lorsqu'il détecte une attaque à un moment donné plutôt qu'à la fin de l'exécution du programme. Si le programme à protéger est un service qui tourne en boucle par exemple, alors le profilage ne sera jamais fait. Ainsi, nous proposons également le mode *échantillonné*, dans lequel nous collectons les informations comportementales en accumulant les valeurs des compteurs à une fréquence prédéterminée. La granularité de l'échantillonnage est la vitesse à laquelle les données des HPC sont collectées. Elle a donc une influence directe sur les performances du processus normal, car elle fait varier le temps d'exécution : un taux d'échantillonnage plus élevé peut éventuellement accélérer la détection, au prix d'un surcoût de performance plus élevé. Dans cette étude, plusieurs granularités sont définies pour les expériences : 100us, 1 ms, 10 ms et 100 ms.

```
Mode complet:
    PAPI_BR_INS,PAPI_TOT_CYC,PAPI_BR_MSP,PAPI_TOT_INS,PAPI_SR_INS,PAPI_LD_INS
    46300,1247387,19130,495840,34407,106392

Mode échantilloné:
    t,PAPI_BR_INS,PAPI_TOT_CYC,PAPI_BR_MSP,PAPI_TOT_INS,PAPI_SR_INS,PAPI_LD_INS
    0,2259,84294,1234,23094,2185,4510
    1,5714,132468,2443,48261,4773,9408
    2,4936,81408,1616,38190,3892,7325
    3,4666,78235,1569,37664,3904,7544
    4,3211,79385,1541,27127,2779,5187
    5,2762,85456,1497,25152,2863,4934
    6,2866,105516,1566,25080,2521,5005
    7,2873,80061,525,32019,2943,6922
    8,3437,77381,845,42811,3352,8516
    9,3119,83830,1491,32169,2733,6209
    10,2667,113056,2199,28530,1549,5118
    11,2428,101686,1876,25065,1475,4646
    12,3585,90674,1581,37589,286,7863
    13,3600,78202,1169,36981,174,7987
    14,3763,79685,1004,34626,2886,7609
    15,3542,138840,2376,33764,2302,6315
    17,4242,96171,939,43190,2842,9771
    18,9329,147598,768,28178,1578,5270
```

FIGURE 4.4 – *Exemple de valeurs HPC dans les modes* complet *et* échantillonné.

Dans la figure 4.4, nous montrons un exemple de valeurs de HPC récupérées dans les deux modes sur un même programme avec une sélection d'évènements arbitraires. Dans le mode *complet*, une seule valeur est récupérée pour chaque HPC à la fin de l'exécution, alors que dans le mode *échantillonné*, nous récupérons (dans cet exemple) 19 valeurs pour chaque compteur pendant l'exécution. Ce sont ces données que nous utilisons ensuite pour entraîner les modèles de ML dans un premier temps, et pour effectuer la classification dans un second temps.

4.2.1.b Entraînement des modèles de Machine Learning

Pour pouvoir classifier le comportement du programme avec un modèle de machine learning, il est nécessaire d'entraîner ce dernier. La deuxième étape consiste alors

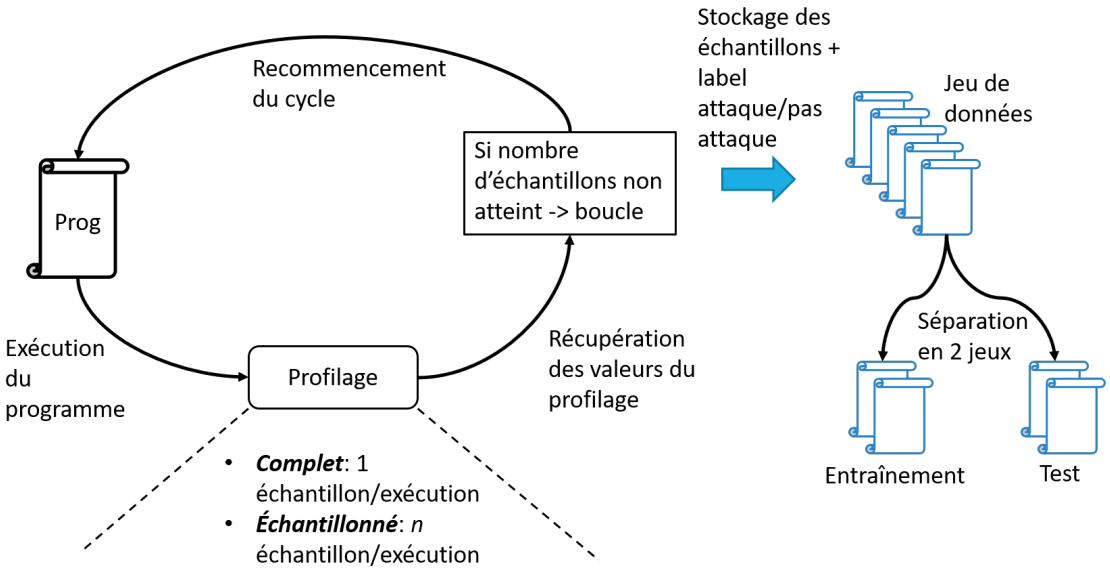

FIGURE 4.5 – *Exemple de construction d'un jeu de données.*

à créer un jeu de données pour entraîner et valider les modèles de ML. Un jeu de données est composé d'échantillons du comportement d'un programme. Dans notre cas, un échantillon est représenté par une série de valeurs récupérées avec les HPC (comme présenté dans la section précédente). La création d'un jeu de données est illustrée par la figure 4.5. Le programme est exécuté et profilé pour extraire les valeurs des HPC. Si le profilage est en mode *complet* alors chaque exécution produit un échantillon. Si le profilage est en mode *échantillonné*, alors chaque exécution produit n échantillons, n dépendant de la durée du programme et de la période d'échantillonnage.

$$n = \frac{durée\_prog}{période\_échantillonnage} \tag{4.1}$$

L'opération est répétée X fois jusqu'à atteindre le nombre d'échantillons voulu dans le jeu de données. À chaque fois qu'un échantillon est ajouté, il est nécessaire de lui attribuer un label "attaque" ou "pas attaque". La particularité du profilage en mode *échantillonné* est que le début de l'exploitation doit être synchronisé avec le processus qui labélise les échantillons puisque toutes les valeurs des HPC collectées ne sont pas de bons ou de mauvais échantillons lors d'une simple exécution. Nous discutons plus précisément de ce cas dans la partie implémentation. Le jeu de données est ensuite découpé en deux jeux de tailles égales, dont un est utilisé pour l'apprentissage (jeu de données d'entraînement) et un pour la validation (jeu de données de test).

De plus, pour que ces ensembles de données soient les plus réalistes possibles, nous définissons également plusieurs charges de fonctionnement sur le système. L'idée est d'introduire une charge de fond à forte intensité de CPU et d'accès mémoire pour prendre en compte le bruit possible du système sur les compteurs [130] et ainsi proposer une solution la plus proche possible de la réalité. Nous utilisons alors des programmes C récupérables sur un git [131]. Nous introduisons trois types de charges :

1. **No Load (NL) :** dans lequel seul le programme est impliqué.

2. **Full Load (FL) :**, où le programme et les benchmarks SPEC sont exécutés. Les conditions FL prennent presque 50% des performances du CPU à elles seules.
3. **Mix Load (MixL) :** un mélange de données FL et NL.

Le cycle de construction d'un jeu de données est donc lancé pour plusieurs types de charges.

4.2.1.c Classification et détection

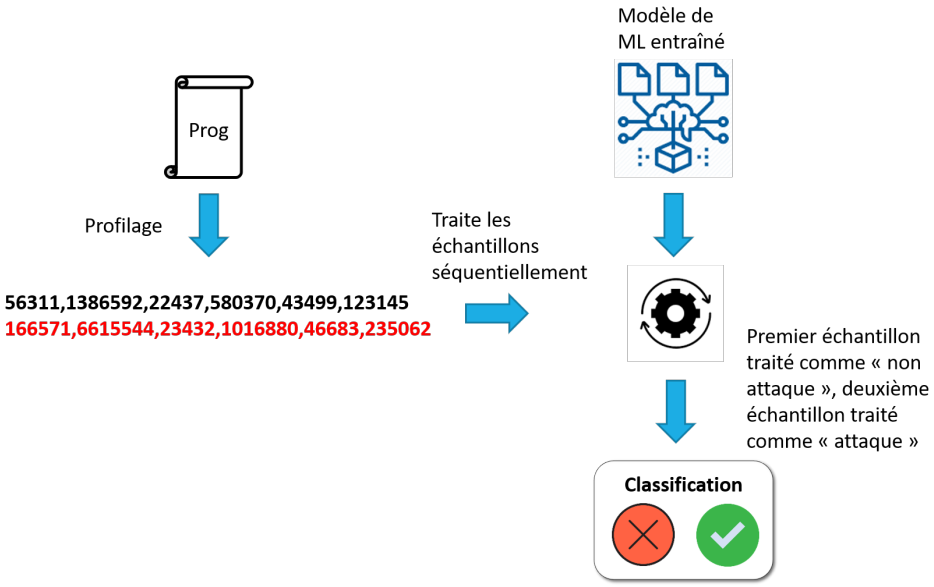

FIGURE 4.6 – *Exemple de classification.*

Dans la dernière phase de la méthode, les modèles entraînés utilisent les valeurs des évènements récupérés à l'aide des HPC pendant l'exécution et classifient chaque échantillon. Le processus est illustré dans la figure 4.6. Nous mettons en scène un exemple avec deux échantillons, un représentant une attaque et un représentant le fonctionnement normal du programme. Le modèle de ML traite ces deux échantillons, indépendamment l'un de l'autre, et prédit si ces derniers sont malicieux. Nous avons vu que la rapidité de détection dépend de la fréquence d'échantillonnage, mais aussi de la rapidité du modèle utilisé. Cette rapidité de détection prend donc deux formes :

1. Rapidité du modèle (en ms) seulement dans le meilleur des cas.

$$R\_Det = R\_modèle. \tag{4.2}$$

2. Rapidité du modèle (en ms) + période d'échantillonnage (en ms) dans le pire des cas.

$$R\_Det = R\_modèle + période\_échantillonnage. \tag{4.3}$$

Les caractéristiques de performance du modèle de ML sont donc à prendre sérieusement en compte lors du choix. Cependant, il n'y a pas que la performance qui rentre en compte, la précision et les taux d'erreurs sont aussi des facteurs déterminants.

Métriques

En machine learning, plusieurs facteurs entrent en compte pour évaluer la précision d'un modèle et dépendent de quatre métriques principales que nous définissons tels que :

- **Les vrais positifs (TP) :** sont des cas dans lesquels un comportement normal est classifié comme tel.
- **Les vrais négatifs (TN) :** sont des cas dans lesquels un comportement d'attaque est classifié comme tel.
- **Les faux positifs (FP) :** sont des cas où une condition de non-attaque est détectée comme une attaque.
- **Les faux négatifs (FN) :** sont des cas où une attaque est signalée comme n'étant pas une attaque.

La précision d'un modèle est considérée comme le taux de réponses correctes, c'est-à-dire :

$$Précision = \frac{TP}{TP + FP} \tag{4.4}$$

Cependant, cette mesure comporte des limites lorsque les jeux de données sont distribués de manière inégale. Prenons un exemple, disons que sur 10 000 échantillons, 1 seul est malicieux, nous avons donc 0.01% de cas d'attaque. Maintenant imaginons que le modèle prédit toujours que le paquet est correct, alors le modèle à une précision de 99.99%, ce qui est très précis. Cependant, il n'est pas capable de détecter la moindre attaque et le taux de TN est de 0%, alors que celui de FN est de 100%. Ce comportement n'est généralement pas acceptable, c'est pourquoi il est important d'évaluer chaque métrique (TN, TP, FP, FN) une à une. Dans notre cas d'usage, nous voulons absolument éviter des faux négatifs, qui représentent une faille dans le système de détection, car l'attaque n'a pas été détectée. Au contraire, les faux positifs peuvent être acceptables, car finalement ils sont considérés comme une fausse alerte. Il ne faut pas que ce chiffre soit trop important cependant, car chaque fausse alerte entraîne une surcharge de performance due au déploiement du rétablissement du système, et une perte de confiance en la solution. Nous proposons alors deux méthodes d'évaluation :

- **à l'aide de la précision** des modèles, en portant une attention particulière aux faux positifs et aux faux négatifs lorsque les données sont distribuées de manière égale entre les cas bons et mauvais.
- **à l'aide de la matrice de confusion**, *i.e.* analyser un à un les taux de succès TN, TP et les taux d'erreurs FP, FN, lorsque les données sont distribuées de manière inégale.

4.2.2 Implémentation du module de détection

Maintenant que nous avons vu la méthodologie derrière le module de détection, nous proposons d'étudier les aspects de l'implémentation. Ces choix peuvent fortement impacter les performances d'un programme, c'est pourquoi nous y portons une attention particulière. Nous proposons de définir dans un premier temps la plateforme utilisée pour implémenter la méthode de protection, puis de détailler l'implémentation pour les trois phases de la méthode:

1. **Profilage à l'exécution**
2. **Entraînement des modèles**
3. **Classification à l'exécution**

4.2.2.a Système

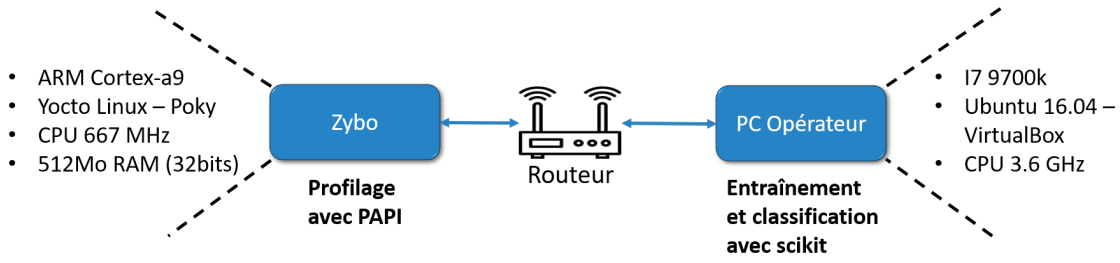

FIGURE 4.7 – *Plateforme pour tester la méthode de détection.*

Pour cette expérience, nous avons réutilisé une partie de la plateforme développée pour la présentation d'attaques au chapitre 2. Nous expliquons cette adaptation à travers la figure 4.7. On retrouve d'un côté la Zybo avec un processeur Zynq-7000 [62] composé de deux CPU ARM cortex-a9 fonctionnant à 667 MHz, avec une distribution poky Linux (Yocto) [64] (noyau 4.19.0-xilinx-v2019.1) et 512 Mo ddr3 (32 bits) de RAM. Tous les programmes profilés sont exécutés sur cette carte. Pour initialiser et récupérer les valeurs des compteurs, il existe de nombreuses bibliothèques et API de haut niveau, à savoir PerfMon [132], OProfile [133], Perftool [134], Intel Vtune Analyzer [135] et PAPI [136], *etc.* Nous utilisons la bibliothèque PAPI [136] pour sa simplicité d'utilisation et son implémentation en C qui permet de la compiler pour plusieurs architectures. De plus, la majorité des processeurs existants sont supportés par la librairie.

En face, on retrouve un PC qui agit comme opérateur du réseau et plus précisément, gère les différentes cartes Zybo déployées. C'est le PC qui est en charge de lancer l'apprentissage sur les cartes et de construire le jeu de données final. Nous utilisons ensuite la librairie scikit-learn [137] version 0.23.0 pour entraîner les différents modèles de ML. Nous utilisons un PC pour cette tâche, car il possède une plus grande puissance de calcul (processeur i7-9700K, tests effectués sur VirtualBox avec un OS Ubuntu 16.04). Il n'est finalement pas intéressant de faire l'entraînement sur la carte Zybo, puisqu'étant moins puissante, nous perdrions seulement du temps. Nous proposons également d'utiliser le PC pour effectuer la classification des échantillons pendant

l'exécution dans l'idée de minimiser les pénalités en performances sur le système embarqué. Néanmoins, les coûts de communication et de bande passante sont à évaluer. Une deuxième raison à ce choix est que, même si l'apprentissage et la classification sont faits sur le PC et non sur la carte Zybo, ils n'affectent pas les résultats finaux de précision. Cela n'a d'incidence que sur le surcoût global de la solution (côté ML) ; nous abordons ce cas dans la section 4.5.

4.2.2.b Implémentation du profilage

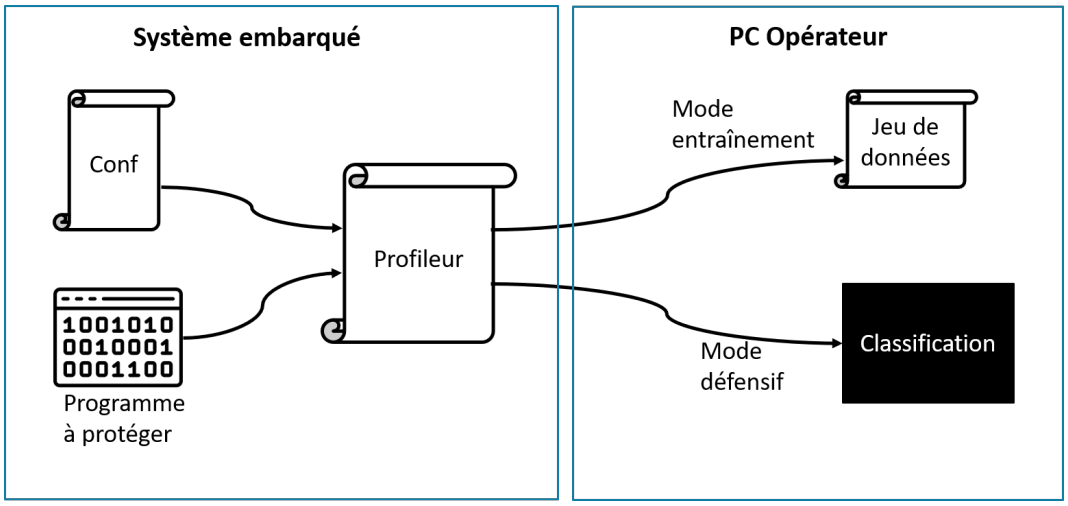

FIGURE 4.8 – *Fonctionnement du profileur.*

La première étape concerne le profilage du programme pour récupérer les valeurs liées aux évènements configurés. Pour collecter les compteurs, nous utilisons la bibliothèque PAPI [136]. C'est une bibliothèque écrite en C, très portable entre les systèmes et très rapide d'exécution. Pour rendre le système facilement évolutif, que ce soit pour un processeur différent, avec ou sans système d'exploitation, nous créons le profilage sous une forme d'encapsulation du programme que nous appellerons **profileur**. Le profileur permet de régler les évènements à tracer d'une part, mais aussi à définir si le programme est exécuté pour créer un jeu de données (mode *entraînement*) ou s'il faut protéger l'exécution (mode *défensif*). La figure 4.8 explique le fonctionnement. Le profileur prend le programme et un fichier de configuration en entrée, et est directement exécuté sur le CPU du système embarqué. Il configure les compteurs, les évènements à tracer, effectue un *fork* du programme et suit ensuite son exécution dépendant du mode choisi par l'utilisateur. Si le mode est réglé sur *défensif*, alors les échantillons récoltés sont classifiés par le PC opérateur et déterminent si le programme est dans une exécution normale ou non. Si le mode choisi est *entraînement*, alors l'exécution du programme est utilisée pour générer un jeu de données qui est ensuite utilisé pour entraîner le modèle de machine learning sur le PC opérateur. Par souci de clarté (code trop long), nous ne pouvons pas présenter de codes sources, nous proposons néanmoins une analyse du pseudocode en mode *entraînement* dans la section suivante, et en mode *défensif* dans la section 4.2.2.d.

4.2.2.c Implémentation de l'apprentissage

L'apprentissage est fait à partir d'un jeu de données. Comme nous protégeons un programme spécifique, l'utilisateur doit générer dans un premier temps des données de l'exécution du programme. Cette partie, si elle est faite à la main, demande énormément de temps et est très répétitive. C'est pourquoi nous avons développé un outil prenant un programme en entrée et générant un jeu de données complet, basé sur les paramètres fournis. Cet outil, écrit totalement en Python, s'exécute sur le PC opérateur. L'idée est que l'utilisateur règle l'environnement en spécifiant le programme sur lequel l'entraînement doit être fait, le mode de profilage (*complet ou échantillonné*), sa fréquence d'échantillonnage dans le cas nécessaire et le nombre d'échantillons requis. La figure 4.9 déroule le fonctionnement. Deux parties sont à prendre en compte:

1. **Le système embarqué** sur lequel le profileur et le programme sont exécutés, représenté par la Zybo.
2. **Le serveur** où l'outil pour générer le jeu de données est lancé par l'opérateur, représenté par le PC.

Le choix technique d'utiliser un serveur pour faire l'apprentissage vient de plusieurs contraintes. D'une part, la taille d'un jeu de données peut vite être conséquente (plusieurs centaines de Mo) et tous les systèmes embarqués ne peuvent pas gérer une telle masse de données. D'autre part, l'exécution de l'outil pour générer le jeu de données sur le système embarqué pourrait le perturber, et ainsi générer du bruit sur les compteurs et fausser l'apprentissage. Enfin, le fait de passer par un serveur permet de paralléliser la génération du jeu de données, c'est-à-dire utiliser plusieurs systèmes embarqués pour générer le jeu plus rapidement.

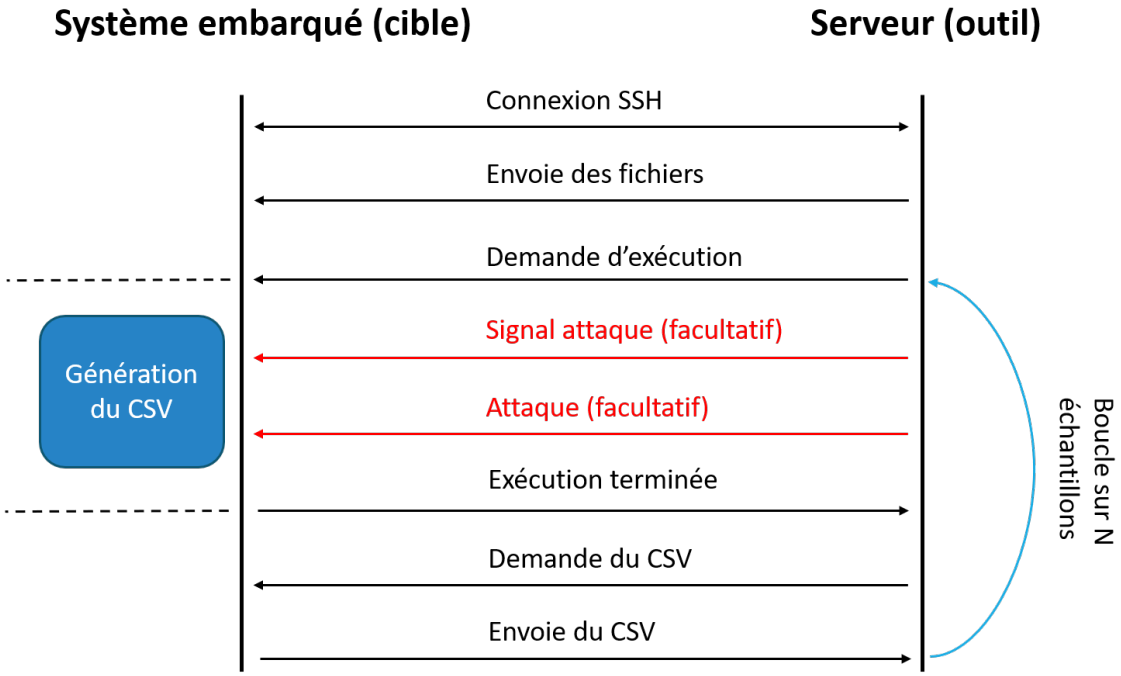

FIGURE 4.9 – *Déroulement de l'outil d'apprentissage.*

Dans un premier temps, le serveur cherche à établir une connexion SSH avec le système embarqué. L'opérateur doit simplement indiquer le programme à exécuter (sur la zybo), le nombre d'échantillons requis, la charge voulue sur le système et le mode (*complet ou échantillonné*). Ensuite, l'outil est lancé (sur le PC), et envoie tous les fichiers nécessaires sur la cible (programme et profileur) et commence l'exécution. Si le profilage est en mode *complet*, alors l'outil attend simplement la fin d'exécution du programme, récupère le fichier CSV généré et labélise l'échantillon. Si le profilage est en mode *échantillonné*, le fichier CSV est rempli au fur et à mesure (dépendant de la période d'échantillonnage), et les labels sont gérés par le profileur. L'outil est donc en charge de signaler au profileur lorsque l'attaque est lancée, pour pouvoir labéliser correctement les échantillons. Enfin, l'opération d'exécution et de récupération du CSV sont répétées N fois, jusqu'à l'obtention du jeu de données complet. Les modèles entraînés sont ensuite générés. Pour mieux comprendre, nous proposons d'étudier le pseudocode du profileur en mode *entraînement* (algorithme 1) et celui de l'outil pour construire le jeu de données puis faire l'apprentissage (algorithme 2).

L'algorithme 1 représente le profileur en mode *entraînement*. Il est composé de deux fonctions principales: *main* et *do_tracing*. La fonction *main* est en charge d'initialiser le fichier d'entraînement (ligne 2), de *fork* le programme (ligne 3). Ainsi dans un cas le programme à protéger est exécuté (ligne 5) et dans l'autre le profilage est lancé (ligne 7 à 10). La fonction *do_tracing* possède deux modes différents: *complet* (ligne 16 à 19) et *échantillonné* (ligne 21 à 34). Dans le mode *complet*, la fonction attend que le programme profilé se termine (ligne 16), et récupère ensuite les valeurs des compteurs (ligne 17) qui sont écrites dans le fichier d'entraînement (ligne 18). Dans le mode *échantillonné*, le déroulement est plus complexe. Dans un premier temps, un serveur socket est lancé (ligne 22). Ce moyen de communication sert à la labélisation des échantillons. Lorsque l'attaque est lancée, il faut en même temps le notifier au profileur. Il rentre ensuite dans une boucle et y reste tant que le programme à protéger n'est pas terminé (ligne 23 à 34). Cette boucle récupère les valeurs des compteurs (ligne 24) et les écrit dans le fichier d'entraînement (ligne 25). Il vérifie ensuite la valeur du label, si ce dernier est toujours égal à 0, alors le profileur attend pour une nouvelle connexion (ligne 26-27). Si une nouvelle connexion est faite, alors le label passe à -1 (ligne 28-30). Sinon le profileur

se met en pause pour le temps défini par *sample_rate* (ligne 32).

Algorithme 1: Pseudocode du profilage en mode *entraînement*.

```
1  Function main(mode,sample_rate or label,program,prog_args):
2      csv_fd ← init_csv() ;
3      pid ← fork() ;
4      if pid == 0 then
5          execve(program, args, NULL) ;
6      else
7          Init_Counters(pid) ;
8          do_tracing(mode, csv_fd, counters, pid, sample_rate or label) ;
9          save(csv_fd) ;
10         clean_counters() ;
11     end
12     return;
13 ;
14 Function do_tracing(mode, csv_fd, pid, sample_rate or label):
15     if mode == complet then
16         wait_while_process_is_running();
17         values ← Get_counters_values() ;
18         csv_fd ← write_values_and_label(values, label);
19         return;
20     else
21         label ← 0 ;
22         sock_fd ← make_socket_server() ;
23         while waitpid(pid, NOHANG) == 0 do
24             values ← Get_counters_values() ;
25             csv_fd ← write_values_and_label(values, label);
26             if label == 0 then
27                 flag ← is_new_connection(timeout=sample_rate) ;
28                 if flag == true then
29                     label ← -1 ;
30                 end
31             else
32                 sleep(sample_rate);
33             end
34         end
35     end
36     return;
```

L'algorithme 2 représente l'outil utilisé pour créer un jeu de données depuis le PC. Nous avons grandement simplifié son implémentation dans l'explication, pour nous concentrer sur le principal en mettant de côté la gestion de plusieurs systèmes embarqués (threads, locks, synchronisations) ou encore la gestion de caches et sauvegardes entre exécutions de l'outil. L'outil prend plusieurs arguments en compte : le mode, le programme à protéger, le taux d'échantillonnage (dépend du mode), l'IP du système embarqué, le nombre d'échantillons requis pour le jeu de données, et l'attaque à

appliquer si nécessaire. La première étape est d'établir la connexion ssh (ligne 2). La deuxième étape consiste à télécharger les fichiers nécessaires (programme à protéger, profileur, configuration) (ligne 3). Ensuite, la charge de fond (condition NL, FL, MixL) est lancée sur le système embarqué (ligne 4). Puis, si le mode de génération est réglé sur *complet*, le déroulement consiste en une boucle, exécutée autant de fois que le spécifie *nb_sample* (ligne 6-15). Cette boucle tire aléatoirement un chiffre (ligne 7) et exécute le mauvais cas ou le bon cas (ligne 9 ou 11). L'idée est de ne pas faire tous les bons cas et tous les mauvais cas d'un coup, pour éviter d'avoir des effets de bord. Une fois l'exécution terminée, le fichier csv est récupéré (ligne 14). En mode *échantillonné* (ligne 17 à 22), le déroulement est presque identique, sauf que cette fois l'outil doit aussi gérer l'attaque et le processus de labélisation (ligne 19). La dernière étape consiste à diviser le jeu de données en deux (entraînement et test) puis à entraîner les modèles de ML sélectionnés (ligne 24 à 26). Une fois le ou les modèles prêts, ils peuvent être utilisés pour classifier les données récupérées des compteurs à l'exécution, en utilisant le mode *défensif* du profileur.

Algorithme 2: Pseudocode de l'outil d'apprentissage.

```
1  Function main(mode, prog_protect, sample_rate, IP, nb_sample, attack, load):
2      ssh_connect_to_device(IP);
3      upload_required_files(prog_protect);
4      fix_load();
5      if mode == complet then
6          for i in range(nb_sample) do
7              rd ← random(0,1);
8              if rd == 0 then
9                  execute_bad_version();
10             else
11                 execute_good_version();
12             end
13             wait_for_end();
14             get_csv();
15         end
16     else
17         for i in range(nb_sample/nb_sample_per_exec) do
18             execute_code();
19             launch_attack_and_notify_profileur();
20             wait_for_end();
21             get_csv();
22         end
23     end
24     create_dataset_train_and_test();
25     do_learning();
26     save_models();
27     return;
28 ;
```

4.2.2.d Implémentation de la classification à l'exécution

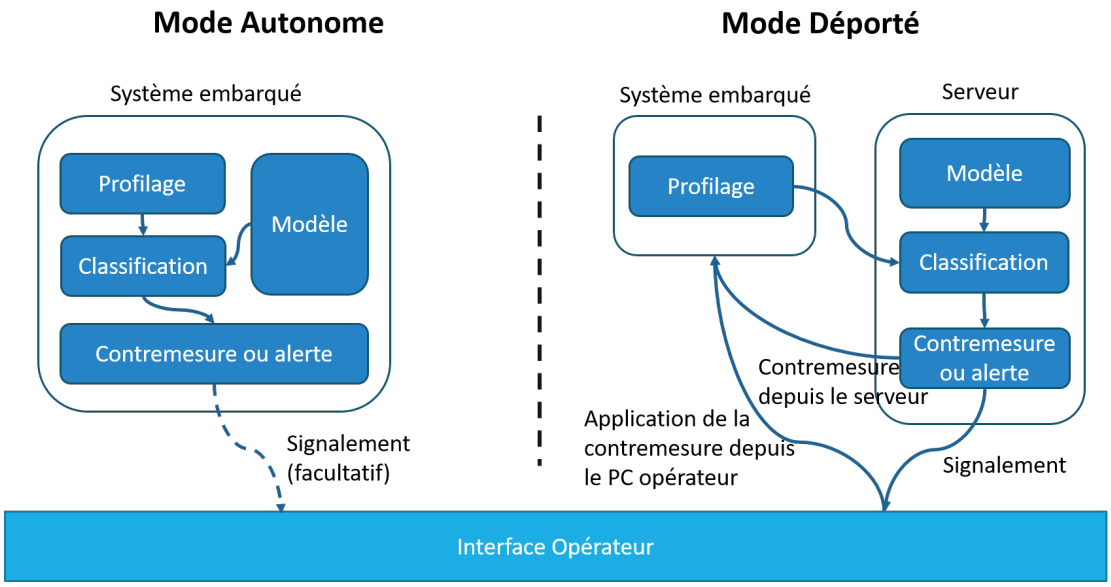

FIGURE 4.10 – *Classification à l'exécution.*

Une fois l'apprentissage terminé, nous pouvons déployer le module de détection sur la cible à protéger. Afin d'obtenir les meilleurs résultats possibles, aussi bien en matière de précision, que de performances, nous définissons deux méthodes de déploiement (figure 4.10):

1. La première méthode, que nous appelons *Méthode autonome*, s'exécute entièrement sur le système embarqué (appelé client) et n'a pas besoin de tierce partie pour fonctionner. L'inférence du modèle est stockée et exécutée sur le système. En cas d'alerte, deux choix sont possibles. Soit le système embarqué est capable de lancer une contremesure pour se défendre seul, soit une alerte est envoyée à l'opérateur pour signaler un problème.

2. La deuxième méthode, *Méthode déportée*, repose sur une détection en deux étapes. L'objet collecte les données des HPC, puis les transmet à un serveur. C'est ce serveur qui est en charge de classifier les échantillons pour déterminer si l'objet est dans un état sûr ou non. Premièrement, le stockage du modèle et son exécution se passent exclusivement sur le serveur, ce qui réduit drastiquement les performances demandées côté client. Deuxièmement, un attaquant avec un accès physique sur l'objet ne peut pas essayer de récupérer les modèles de prédiction pour les étudier et essayer de les contourner. Cette méthode implique donc une plus forte sécurité des modèles en plus d'une baisse de performance côté client. D'un autre côté, la solution demande une plus grande bande passante pour envoyer les échantillons à classifier. Selon le taux d'échantillonnage, cela peut être problématique. De plus, le temps de prédiction d'une attaque dépend aussi de cet aspect communication. En effet, là où l'attaque est détectée directement après la classification côté client, en mode *déporté* il faut prendre en compte le temps de la communication (aller/retour). Cependant le modèle s'exécute plus vite sur le serveur que sur l'objet (processeur plus puissant), cette perte de temps est donc à

évaluer. Enfin, cette méthode présente l'avantage de mutualiser les modèles sur une flotte d'objets et d'ainsi affiner l'apprentissage à l'exécution et de laisser à l'opérateur un champ d'action plus large vis-à-vis de sa flotte, scénario qui se prête très bien à l'utilisation du LwM2M.

Algorithme 3: Module de détection à l'exécution (profileur en mode *défensif*) - côté client.

```
   Input: sample_rate,program,ServerAddress
 1 if ServerAddress then
 2 |   client ← Connect_to_server(ServerAddress) ;
 3 end
 4 pid ←Fork() ;
 5 if pid == 0 then
 6 |   execve(program, args, NULL) ;
 7 else
 8 |   Init_Counters(pid) ;
 9 |   while Process is running do
10 |   |   values ← Get_counters_values() ;
11 |   |   if client then
12 |   |   |   Send_counters_values(client, counters_values) ;
13 |   |   else
14 |   |   |   alert ← ML_Classifier(counters_values) ;
15 |   |   |   if alert then
   |   |   |   |   /* Attack Detected                    */
16 |   |   |   |   return 1 ;
17 |   |   |   end
18 |   |   end
19 |   |   Sleep(SamplingRate) ;
20 |   end
21 |   Clean_counters() ;
22 end
23 return 0 ;
```

Pour mieux comprendre les méthodes, nous expliquons ces deux mécanismes à l'aide d'algorithmes. L'algorithme 3 présente le pseudocode du module de détection côté embarqué (représentant le profileur en mode défensif). Comme décrit, le module prend en entrée le `SamplingRate` (taux d'échantillonnage), qui peut être défini par l'utilisateur ou modifié au moment de l'exécution pour contrôler efficacement les pénalités de performance. Une autre entrée est `ProcesstoProtect`, qui fait référence au programme à protéger. Enfin, il y a `ServerAddress`, qui permet de configurer un serveur pour le client dans le cas où le mode défensif réglé est *déporté*. Si aucune adresse n'est donnée, alors le mode reste sur *autonome* et le module utilise le modèle de machine learning présent sur le système embarqué pour classifier les échantillons et sécuriser ce dernier. Pour rentrer un peu plus dans le détail de l'algorithme, les lignes 1 à 3 permettent de configurer le client pour qu'il se connecte au serveur dans le cas où une adresse est donnée. La ligne 4 *fork* le programme à protéger, c'est-à-dire que le module obtient son PID pour configurer les compteurs, et commence à exécuter le

programme en parallèle. Les compteurs sont ensuite configurés (ligne 8). De la ligne 9 à 20, c'est une boucle jusqu'à ce que l'exécution du programme soit terminée. Les valeurs des compteurs sont récupérées à la ligne 10. Si un serveur est configuré, alors les données lui sont envoyées (ligne 11-13), sinon les lignes 14 à 17 sont exécutées. La ligne 14 traite la classification et les lignes 15-17 indiquent si l'attaque est détectée ou non. Il retourne ensuite en mode veille (ligne 19) et exécute à nouveau la boucle une fois la veille terminée (sur la base du taux d'échantillonnage).

Algorithme 4: Module de détection à l'exécution - côté serveur.

```
   Input: ServerAddress
1  server ← Init_server(ServerAddress) ;
2  while True do
3  |  counters_values ← Wait_data_from_client(server) ;
4  |  alert ← ML_Classifier(counters_values) ;
5  |  if alert then
   |  |  /* Attack Detected                                    */
6  |  end
7  end
8  return 0 ;
```

Dans l'algorithme 4, nous présentons le programme côté serveur, dans le cas où l'inférence du modèle est exécutée sur le serveur. L'algorithme 4 prend en compte `ServerAddress` comme entrée et initialise le serveur à la ligne 1. Il entre ensuite dans une boucle infinie (ligne 2-7). Les valeurs des compteurs sont reçues à la ligne 3. Après la collecte des données, le programme les classifie en utilisant le modèle de ML pré-entraîné (ligne 4) et signale une erreur si la prédiction lève une alerte lignes 5-6. Du moment que tous les objets qui s'enregistrent auprès du serveur exécutent le même code, alors le serveur est capable de traiter la classification pour tous et d'indiquer à l'opérateur quels sont les objets qui fonctionnent correctement et ceux qui sont sous attaque.

Dans les résultats, nous présentons les expérimentations avec le mode *déporté*. Les raisons derrière ce choix sont simples. D'une part, quel que soit le mécanisme utilisé, les résultats de précision et de la matrice de confusion ne changent pas. Seuls les résultats de surcoût de performance varient (partie sur le machine learning). D'autre part, à des fins de tests et de validations de la méthode globale, faire l'inférence en C d'un modèle de ML est une tâche d'ingénierie complexe et coûteuse. C'est pourquoi nous avons choisi de valider la méthode à travers des expériences et des résultats sur la version *déportée* qui permet l'utilisation de bibliothèques très utilisées comme scikit [137].

4.3 Sélection des caractéristiques

Maintenant que l'outil de détection est implémenté et fonctionnel, il reste à sélectionner les évènements d'une part et les modèles de machine learning d'autre part. Nous proposons alors de définir des scénarios d'usage pour permettre d'évaluer et d'expérimenter ces choix.

4.3.1 Définition des scénarios

Dans cette étude, nous définissons trois scénarios, pour lesquels nous allons créer un jeu de données, afin d'évaluer les choix techniques de notre solution. L'idée est d'utiliser ces scénarios pour valider le choix des évènements, des modèles de machine learning et d'implémentation pour garder une solution la plus simple et efficace possible avec le moins de pénalités. Nous présentons ces trois scénarios dans le tableau 4.1.

Tableau 4.1 – *Liste de scénarios utilisés.*

Scénarios	**Description**
Scénario 1 & Jeu de données 1	• Généré à partir du programme 1 en annexe. • Le programme contient une vulnérabilité seulement. • Profilage en mode *complet*. • Contient 20,000 échantillons.
Scénario 2 & Jeu de données 2	• Généré à partir du programme 2 en annexe. • Le programme contient une vulnérabilité et une attaque. • Profilage en mode *complet*. • Contient 20,000 échantillons
Scénario 3 & Jeu de données 3	• Généré à partir du programme 3 en annexe. • Le programme contient une vulnérabilité et plusieurs attaques. • Profilage en mode *échantillonné*. • Contient 1,000,000 échantillons

Scénario n°1.

Il concerne le programme 1 en annexe (A). Le programme déclare un tampon de 10 octets, et laisse choisir à l'utilisateur un index du tableau pour y écrire la valeur 1. Cependant, le programme ne vérifie pas que l'index ne dépasse pas la taille du tampon, ainsi il y a une vulnérabilité, et l'utilisateur peut écrire au-delà de la limite du tampon. Ce scénario ne contient pas d'attaque, l'idée est de tester si à partir des valeurs

des compteurs et d'un modèle de ML, il est possible de détecter une vulnérabilité non exploitée dans le code. La classification propose alors deux choix, vulnérable/non vulnérable. Nous utilisons le profilage en mode *complet* pour générer un jeu de données de 20 000 échantillons, avec une répartition de 10 000 en fonctionnement vulnérable et de 10 000 en fonctionnement normal. Le choix du nombre d'échantillons est arbitraire, mais quelques contraintes se sont tout de même posées. Il fallait assez d'échantillons pour pouvoir faire un apprentissage, mais chaque exécution d'un programme prend un temps non négligeable. Admettons qu'il faut 10 secondes pour générer un échantillon (lancement du programme, entrée utilisateur, exécution du programme, récupération d'un échantillon, mettre un label, stocker l'échantillon, recommencer), alors la construction du jeu de données prend déjà plus de deux jours. Bien qu'il soit possible de réduire cette durée en parallélisant la génération d'échantillons, dans un premier temps, nous souhaitions garder un ensemble de données de taille relativement faible pour ajuster le choix d'évènement/modèle de ML.

Scénario n°2.

Le jeu de données est construit à partir du programme 2 en annexe (A). Ce programme prend aussi une entrée utilisateur qui définit deux cas :

1. Un cas normal de fonctionnement.
2. Un cas ou le programme s'applique lui-même un dépassement de tampon sur la pile.

Dans les deux cas de fonctionnement, le programme définit deux tampons, et copie les mêmes données dans ce tampon. Cependant, dans un cas le tampon utilisé est trop petit, et lors de l'écriture, la sauvegarde du pointeur de retour est écrasée, et appelle la fonction *hijack()* au lieu de retourner dans *main()*. La fonction *hijack()* est en fait notre comportement d'attaque directement embarquée dans le programme pour plus de simplicité de lancement et de rapidité d'exécution (génération de jeu de données plus rapide). Elle consiste à ouvrir un fichier, le lire, et l'afficher. Il y a plusieurs cas typiques d'attaques, l'ouverture d'un shell, l'écriture dans un fichier, la lecture d'un fichier, ou encore l'ouverture d'une connexion réseau. Il est très complexe de faire un choix, car les possibilités d'attaques et de méthodes sont infinies (aussi bien au niveau exploitation que post exploitation). Néanmoins, il est nécessaire d'avoir un scénario d'attaque au moins pour valider la méthode, ici nous avons fait le choix d'une lecture de fichier. Ce jeu de données est aussi construit à partir d'un profilage en mode *complet* et contient 20 000 échantillons avec une répartition de 10 000 en fonctionnement normal et 10 000 en fonctionnement attaque.

Scénario n°3.

Le jeu de données est construit à partir du programme 3 en annexe (A). Cette fois, le programme est un service qui tourne en boucle. C'est un serveur *socket*, qui attend une

connexion, et renvoie simplement les messages à l'expéditeur. Le programme contient une vulnérabilité de formatage de chaîne de caractères, qui permet à un utilisateur malintentionné de lancer une attaque lorsqu'il est connecté. Le but de ce scénario est de tester la méthode de détection en utilisant le profilage en mode *échantillonné*. Nous définissons le code d'exploitation, écrit en python, dans l'annexe "exploitation du programme 3" (A). Cette attaque à deux rôles :

1. Le premier consiste à générer du trafic aléatoire (nombre de paquets et contenu aléatoire) sur le réseau pour que le code ait le temps de générer des échantillons du comportement "normal" qui sont labélisés comme bon.
2. Une fois cette séquence terminée, le code d'exploitation exploite la vulnérabilité du *sprintf* et écrase l'entrée GOT de la fonction *strncmp* pour la remplacer par la fonction *system*. Ainsi, il obtient un shell de commande (cas classique d'exploitation). Les échantillons générés pendant l'attaque sont labélisés comme mauvais.

Ici, 1 000 000 d'échantillons sont générés. Cette fois, nous n'avons pas opté pour une distribution 50% échantillons mauvais/50% échantillons bons. Il y a plusieurs raisons, discutables, à ce choix. Le premier est qu'il n'est finalement pas réaliste d'avoir des jeux de données avec des distributions 50/50 sur les comportements attaque/non attaque, car en production, le système est la plupart du temps dans des conditions normales d'exécution. Deuxièmement, et pour la même raison que déjà expliquée, il faut prendre en compte le temps de génération d'un jeu de données. À la différence du mode *complet*, en mode *échantillonné*, beaucoup d'échantillons sont générés sur une seule exécution, mais la durée de l'attaque ne concerne qu'environ 5% du temps d'exécution. Dans ces conditions, pour générer un jeu de données de 1 000 000 d'échantillons, il faut déjà compter presque une semaine d'exécution complète. Même si cette génération peut être parallélisée, il faut prendre en compte ce temps lors des tests et du développement de la méthode. Il devient donc compliqué d'avoir une répartition égale. Nous proposons alors une distribution d'environ 95% de bons échantillons et 5% de mauvais échantillons. De plus, cette distribution inégale aidera les modèles de ML à mieux s'entraîner sur le comportement normal du programme que sur le comportement d'attaque qui n'est finalement pas connu.

4.3.2 Compteurs de performance

Les compteurs de performance matériels (HPC) sont des registres spéciaux qui sont présents dans presque toutes les familles de processeurs contemporains (ARM, Intel, etc.). Nous utilisons ces compteurs pour récupérer des évènements concernant le comportement du programme et ainsi créer un modèle de ce dernier. Ces évènements peuvent être, par exemple, des accès manqués au cache, le nombre total de cycles du processeur pour exécuter le programme, ou encore le nombre d'instructions de branchement effectué. Toutefois, les types de compteurs matériels varient d'une architecture à l'autre en raison de la diversité des organisations matérielles. De plus, le nombre limité de compteurs pour stocker les évènements oblige souvent les utilisateurs à effectuer plusieurs mesures pour collecter toutes les caractéristiques souhaitées.

Tableau 4.2 – *Liste des évènements matériels disponible dans les processeurs ARM.*

No.	Hardware Event as Features	Feature ID
1	Level 1 Data Cache Misses	L1_DCM
2	Level 1 Instruction Cache Misses	L1_ICM
3	Data Translation Look Aside Buffer Misses	TLB_DM
4	Instruction Translation Lookaside Buffer Misses	TLB_IM
5	Hardware Interrupts	HW_INT
6	Conditional Branch Instructions Mispredicted	BR_MSP
7	Instructions Issued	TOT_IIS
8	Instructions Completed	TOT_INS
9	Floating Point Instructions	FP_INS
10	Load Instructions	LD_INS
11	Store Instructions	SR_INS
12	Branch Instructions	BR_INS
13	Vector/SIMD Instructions	VEC_INS
14	Total Cycles	TOT_CYC
15	Level 1 Data Cache Accesses	L1_DCA

Bien que les HPC soient présents dans tout type de processeur, et que les concepts restent les mêmes, nous avons choisi de focaliser cette étude sur l'architecture ARM. Nous gardons le processeur Zynq-7000 qui possède deux CPU cortex-a9 [63] et tous les détails donnés dans ce chapitre sur les évènements et les compteurs matériels sont pour ce type de cortex. Le cortex-a9 permet de configurer 58 évènements différents en utilisant les HPC. Cependant, il n'y a que 6 compteurs matériels disponibles, de sorte que la surveillance de plus de 6 évènements donne lieu à un multiplexage de ces derniers. Ils perdent de ce fait en fiabilité, car un seul compteur agrège des données pour plusieurs évènements.

De plus, il existe deux catégories d'évènements dans un processeur :

1. Les évènements **natifs** qui sont spécifiques au processeur dans lequel ils se trouvent.

2. Les évènements **prédéfinis** qui sont des évènements génériques que l'on peut retrouver dans tout type de processeurs.

Nous choisissons alors de ne travailler qu'avec les évènements prédéfinis pour avoir une solution adaptable et réduisons donc la liste à 15 évènements. Le tableau 4.2 présente cette liste d'évènements matériels prédéfinis auxquels on peut accéder sur le processeur Zynq-7000. Néanmoins, même s'il ne reste plus que 15 évènements, choisir un maximum de 6 évènements utiles à la caractérisation du système est une tâche relativement complexe. C'est pourquoi, dans la section 4.3.2.a, nous fournissons le raisonnement de la sélection d'un ensemble d'évènements matériels appropriés pour notre outil de détection.

4.3.2.a Sélection des évènements liés aux compteurs de performances matériels

La sélection des meilleurs évènements pour la caractérisation du système est un problème complexe. Chaque programme est différent, que ce soit dans ses fonctionnalités, dans son comportement ou même dans son fonctionnement sur le matériel. Certains programmes ont un flot d'exécution très simple et utilisent beaucoup de données alors que d'autres ont un flot d'exécution complexe, mais n'utilisent presque pas de données. De plus, une contrainte très forte s'ajoute à cette mixité de comportement: l'attaque à détecter est inconnue. Il devient donc très complexe de mettre en place une analyse formelle ou théorique, puisque les données en entrée du problème sont inconnues. Ces particularités font que pour choisir la meilleure combinaison d'évènements possible, et pour qu'elle soit réutilisable quel que soit le programme qui s'exécute, il faut appliquer une méthode empirique. Pour cela, nous avons principalement utilisé deux approches :

1. La première approche consiste à raisonner sur l'utilité globale de l'évènement pour caractériser le système. Est-ce que l'évènement est pertinent quel que soit le code ? Est-ce qu'il donne des informations sur l'exécution d'un code différent ? À quel point l'information est pertinente ? Les réponses à ces questions fournissent une présélection. Même si l'on pourrait imaginer utiliser des évènements différents selon le programme à protéger, nous sommes partis de l'hypothèse d'avoir dans un premier temps une sélection standard, utilisable pour tout programme. Si cette sélection standard donne de bons résultats, on peut facilement imaginer qu'avec encore plus d'analyse sur les évènements et une sélection dépendant du programme à protéger, alors la précision de détection sera encore meilleure.

2. La seconde approche est de tester de manière empirique, parmi les évènements restants, ceux qui fournissent les meilleures informations sur le comportement du système pour arriver à une liste de 6 évènements maximum. Ici, l'idée est d'adopter une méthode exhaustive sur les combinaisons d'évènements pour avoir les meilleurs résultats possibles quant au choix des évènements. Nous testons les combinaisons avec la première liste de modèles de ML (tableau 4.6) en observant les précisions moyennes des modèles (usage par défaut).

Pour étayer le raisonnement de la première partie, nous utilisons le programme 3 présenté en annexe (A) et nous récupérons des valeurs d'évènements à intervalle régulières (1ms). L'idée est de montrer l'impact qu'a l'exécution d'un nouveau code (à travers une attaque par corruption mémoire) sur les compteurs. Cette méthode n'apporte en rien une preuve formelle sur l'utilité des compteurs, mais les contraintes au problème font qu'il est impossible de se fixer sur l'utilisation précise d'un programme ou d'une attaque pour une analyse plus formelle. Les possibilités sont infinies. Ici, l'idée est donc de regarder tous les évènements un à un, à travers un exemple, pour vérifier si nos hypothèses et intuitions le concernant se confirment.

L'évènement HW_INT, qui représente une interruption venant du matériel, peut avoir des relations avec l'exécution d'un nouveau code, par exemple une routine d'interruption qui lance des actions spécifiques. Cependant ce n'est pas un cas typique

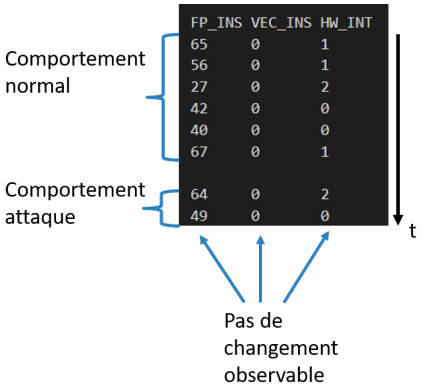

FIGURE 4.11 – *Exemple de valeurs HPC pour FP_INS, VEC_INS et HW_INT.*

dans tout programme. Deux autres évènements sont très liés au type de code qui s'exécute : FP_INS qui représente le nombre d'instructions à virgule flottante et VEC_INS qui représente le nombre d'instructions VectorSIMD. Cette particularité implique que ces caractéristiques peuvent ne pas représenter le comportement de certains programmes et donc ne nous oriente pas vers une combinaison d'évènements standard. Comme on peut l'observer dans l'exemple proposé (figure 4.11, les valeurs récupérées ne changent pas vraiment entre l'exécution normale du programme de référence et l'exécution de l'attaque.

On trouve ensuite les évènements liés à la mémoire cache de niveau 1 :

- L1_ICM concerne les accès manqués au cache d'instructions.
- L1_DCM est lié aux accès manqués des données.
- L1_DCA concerne les accès aux données dans la mémoire cache.

Ces évènements sont de bons candidats pour une combinaison standard. Si un nouveau code est exécuté, alors il est possible que les instructions et données (du programme) utilisées jusqu'à présent changent, et ainsi impactent les accès manqués au cache.

Les évènements TLB_DM et TLB_IM représentent des erreurs de cache de données et d'instructions dans le *Translation Lookaside Buffer* (TLB) utilisé par l'unité de gestion de la mémoire (MMU), et sont davantage liés à des gestions d'adresses et pages virtuelles qu'à une nouvelle exécution de code dans un processus existant. Ces évènements ont cependant tendance à apparaître lors de l'utilisation d'appels système par exemple. Cependant, même si une attaque contient souvent des appels système, si le code protégé contient aussi des appels système, alors il n'est pas forcément possible de distinguer un bon comportement d'un mauvais. Ils font donc des candidats modérés, et les évènements liés au cache de niveau 1 semblent plus déterminants. On retrouve un facteur entre 2 et 4 pour les évènements liés au cache de niveau 1 entre le comportement d'attaque et le comportement normal alors que ce facteur n'est même pas de 2 pour les TLB (figure 4.12).

LD_INS et SR_INS représentent les instructions de chargement et de stockage des

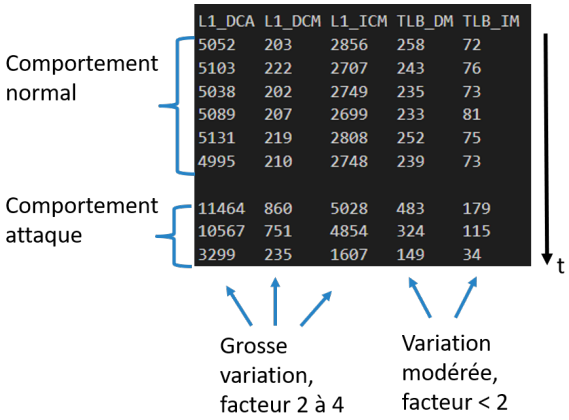

FIGURE 4.12 – *Exemple de valeurs HPC pour mémoire cache et TLB.*

données du programme dans la mémoire. Si un code différent s'exécute, il est fortement possible que ces valeurs changent. Par exemple si on est sur une application qui utilise majoritairement un flot de données (type crypto par exemple), alors, lors d'une attaque le nombre de *load* et *store* diminuera probablement en conséquence. On peut observer (sur la figure (4.13) une variation parfois forte, parfois plus faible (d'un facteur 10 à une variation nulle). C'est donc un indicateur potentiel, intéressant à coupler avec d'autres informations, pour détecter une anomalie dans le comportement normal du programme.

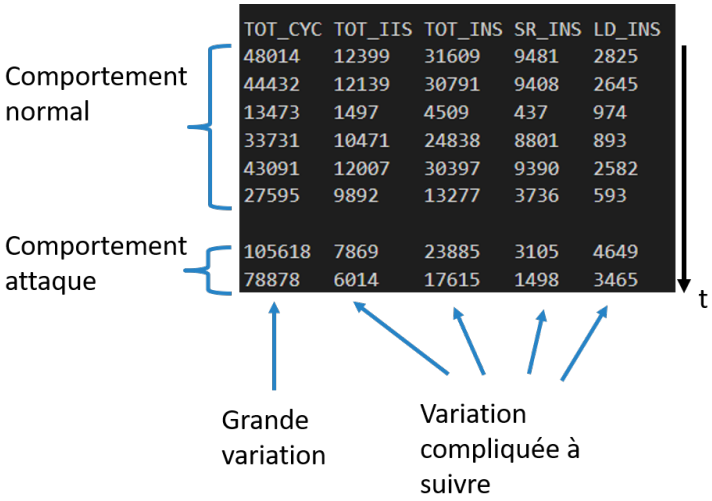

FIGURE 4.13 – *Exemple valeurs HPC pour les évènements de charge.*

TOT_IIS, TOT_CYC et TOT_INS représentent respectivement le total des instructions émises, le nombre de cycles total et le total des instructions terminées. La charge du CPU a un impact direct sur ces évènements ; ainsi, l'exécution d'un nouveau code (ou différent) a également un impact sur ces évènements. Nous montrons un exemple de l'impact dans l'image 4.13. Les mêmes conclusions sont constatables que pour les instructions de *load* et *store*. Une variation est observée au moment de l'attaque, mais difficilement déterminable, notamment pour les évènements TOT_CYC et TOT_INS. Ces trois évènements peuvent donc être des signaux intéressants pour la détection d'attaques.

Enfin, il y a BR_MSP et BR_INS. Le premier représente une mauvaise prédiction

de branchement, tandis que le second représente le nombre d'instructions de branchement effectuées. Ces deux évènements donnent beaucoup d'informations sur le code actuellement exécuté. Ici, il n'est pas nécessaire de donner un exemple pour comprendre que le nombre d'instructions de branchement peut varier, et que la prédiction erronée de branchement va également varier lorsqu'un code différent s'exécute. Ces deux derniers évènements sont donc un très bon indicateur d'exécution de code différent.

Tableau 4.3 – *Comparaison des évènements.*

No.	Feature ID	Pertinence supposée
1	L1_DCM	++
2	L1_ICM	++
3	TLB_DM	+
4	TLB_IM	+
5	HW_INT	-
6	BR_MSP	+++
7	TOT_IIS	++
8	TOT_INS	++
9	FP_INS	-
10	LD_INS	++
11	SR_INS	++
12	BR_INS	+++
13	VEC_INS	-
14	TOT_CYC	++
15	L1_DCA	++

Sachant que le nombre d'évènements à choisir est limité (max 6), nous devons faire des choix. À la deuxième étape, nous rencontrons un problème de puissance de calcul pour tester toutes les combinaisons pour X éléments. En effet, le calcul de la précision moyenne de tous les modèles de ML sélectionnés prend environ 10 secondes. Avec 9 éléments, il y a 9! possibilités, soit 362 880 combinaisons. Il faudrait donc 42 jours pour faire ce traitement. En enlevant un évènement, il reste 8! possibilités, soit 40 320 combinaisons. La durée est réduite à environ 4 jours, ce qui reste acceptable. Nous devons donc présélectionner 8 évènements maximum pour passer à la phase 2. Sur la base du raisonnement effectué, il est donc nécessaire de supprimer 7 évènements. Pour cela, nous leur avons attribué une note en fonction de la pertinence évaluée, du raisonnement effectué, des conclusions émises et de la reproductibilité sur d'autres programmes (tableau 4.3). Nous choisissons d'enlever tous les évènements avec un score "-" et "+" (total de 5). Pour éliminer les 2 derniers, nous proposons d'enlever ceux qui pourraient être redondants parmi les scores "++". Il y a 3 évènements basés sur les caches, deux représentant des accès manqués et un des accès seulement. Nous faisons donc le choix d'enlever L1_DCA. Même raisonnement pour les évènements de charge CPU, on enlève TOT_IIS. Nous obtenons donc une liste de 8 évènements, présentée dans le tableau 4.4.

Nous avons ensuite testé toutes les combinaisons d'évènements possibles. Nous avons utilisé le programme 2 présenté en annexe (A) et récolté 20 000 échantillons en mode *complet* et 20 000 en mode *échantillonné* (1ms) pour chaque évènement. Il y

Tableau 4.4 – *Liste des évènements présélectionnés.*

No.	Nom de l'évènement	ID de l'évènement
1	Conditional Branch Instructions Mispredicted	BR_MSP
2	Branch Instructions	BR_INS
3	Instructions Completed	TOT_INS
4	Total Cycles	TOT_CYC
5	Store Instructions	SR_INS
6	Load Instructions	LD_INS
7	Level 1 Data Cache Accesses	L1_ICM
8	Level 1 Data Cache Misses	L1_DCM

a une répartition égale entre les échantillons corrects et les échantillons d'attaque. Le choix de ce programme pour tester les combinaisons peut être discutable encore une fois. Nous avons volontairement sélectionné un programme simple pour ne pas être impacté par un programme avec une trop forte signature sur le flot de contrôle ou l'utilisation de données et aurait ainsi pu biaiser notre sélection standard. Chaque jeu de données ainsi créé a été testé avec plusieurs modèles de ML (réglages par défaut) pour déterminer une moyenne de précision. Par souci de clarté, au vu de toutes les combinaisons testées, nous ne pouvons pas présenter tous les résultats de précision. Nous avons obtenu la combinaison présentée dans le tableau 4.5.

Tableau 4.5 – *Combinaison d'évènements sélectionnée.*

No.	Nom de l'évènement	ID de l'évènement
1	Instructions Completed	TOT_INS
2	Total Cycles	TOT_CYC
3	Store Instructions	SR_INS
4	Load Instructions	LD_INS
5	Conditional Branch Instructions Mispredicted	BR_MSP
6	Branch Instructions	BR_INS

4.3.3 Modèles de Machine Learning (ML)

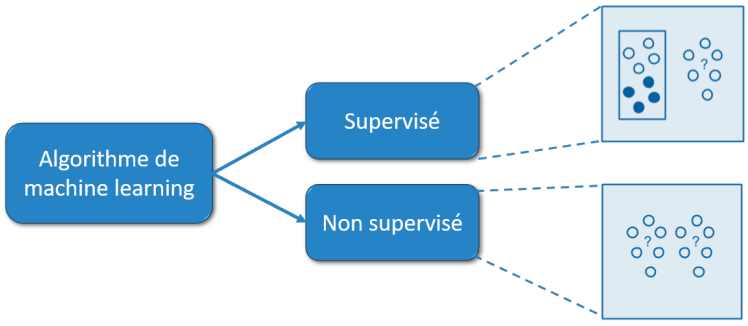

FIGURE 4.14 – *Exemple de types d'apprentissage pour les algorithmes de machine learning.*

Dans cette section, nous débattons de la manière dont les modèles de ML peuvent être utiles à la classification des échantillons. Comme nous l'avons vu dans la section

4.3.2, les HPC fournissent indirectement des informations sur l'exécution d'un programme. Dans l'état de l'art, nous avons vu que les mécanismes à base de seuils de statistiques ne sont pas suffisants pour traiter de manière optimale toutes les données des HPC. Cependant, le machine learning peut aider à classifier des problèmes à l'aide de jeux de données utilisés pour l'apprentissage. Cet apprentissage repose sur un jeu de données collecté au préalable et peut prendre plusieurs formes: supervisé, non supervisé, semi supervisé, *etc*. Les deux modes les plus connus sont supervisés et non supervisés (voir figure 4.14). Dans le premier cas, l'apprentissage est fait en fournissant au modèle de ML les solutions pour chaque cas possible. C'est-à-dire que si il y a deux sorties de classification possibles, par exemple bon et mauvais, alors il faut indiquer pour chaque échantillon qui est bon et qui est mauvais (processus de labélisation). Dans le cas des modèles non supervisés, l'apprentissage est beaucoup plus simple, il n'est pas nécessaire d'expliciter les labels des échantillons, tout est automatiquement classifié. C'est une fonctionnalité intéressante dans notre cas d'étude, car il ne serait plus nécessaire de labéliser chaque échantillon qui est une tâche complexe à synchroniser (entre le moment où le programme s'exécute normalement et le moment de l'attaque).

FIGURE 4.15 – *Catégories d'algorithmes de machine learning (Crédit Jason Brownlee [1]).*

Quel que soit le mode d'apprentissage sélectionné, il existe une multitude d'algorithmes de machine learning (voir figure 4.15), découpée en plusieurs catégories. L'idée a été de présélectionner un ou deux modèles pour chaque catégorie d'algorithmes pertinente pour faire de la détection d'anomalies. Les critères à respecter sont : un modèle rapide, léger et capable de faire de la classification. Nous proposons de tester plusieurs types de modèles même si l'intuition de départ quant à leurs fonctionnalités semble désavantageuse. L'étape finale de la sélection consiste à affiner ces choix, et voir si les intuitions sont validées à travers une expérimentation.

4.3.3.a Sélection des modèles de Machine Learning

Pour commencer la présélection, nous partons de la cartographie des modèles de ML proposée par Jason Brownlee [1] (figure 4.15) et nous observons les particularités de chaque catégorie pour les modèles supervisés :

- **Les algorithmes de régression :** la régression est une méthode d'analyse statistique permettant de trouver une relation entre des variables qui sont corrélées. C'est une méthode de classification, dont les modèles les plus connus sont la régression linéaire et la régression logistique. L'idée de régression linéaire consiste simplement à trouver une droite qui correspond le mieux aux données, *i.e.* s'approche de l'idée d'un seuil. Ces modèles semblent donc limités pour notre cas d'usage, mais leur faible coût peut en faire un candidat intéressant. Nous sélectionnons donc ici le modèle de régression logistique (LR), qui est basée sur la même idée qu'une régression linéaire, mais pour faire de la classification.

- **Les algorithmes basés sur les instances :** Ce type d'algorithme construit des hypothèses directement depuis les données d'entraînement. Ces méthodes consistent généralement à constituer une base de données d'exemples et à comparer les nouvelles données à la base de données en utilisant une mesure de similarité afin de trouver la meilleure correspondance et de faire une prédiction. De ce fait, les modèles sont généralement lourds (coût mémoire conséquent). La complexité de calcul est O(n) dans le pire des cas, n étant le nombre d'échantillons d'apprentissage. Le temps de prédiction augmente donc proportionnellement à la taille du jeu de données d'apprentissage. Cette caractéristique peut être problématique, mais l'un des avantages de l'apprentissage basé sur des instances par rapport aux autres méthodes d'apprentissage automatique est sa capacité à adapter son modèle à des données encore jamais vues (détection d'anomalies). C'est cette fonctionnalité qui nous laisse une intuition intéressante vis-à-vis de ce modèle. De plus, la complexité de l'algorithme peut être réduite en utilisant seulement k correspondances parmi n, $k \ll n$. Nous sélectionnons donc deux modèles de cette catégorie: K-Nearest Neighbors (KNN) et Support Vector Machine (SVM).

- **Les algorithmes de régularisation :** Ce type de modèle est une extension des modèles de régression qui pénalise les modèles en fonction de leur complexité, en favorisant des modèles simples. Ce scénario n'entre pas dans notre cas d'usage, nous n'avons donc pas sélectionné d'algorithme de ce type.

- **Les algorithmes d'arbres de décision :** Les méthodes d'arbre de décision construisent un modèle de décision à partir des données d'apprentissage et font une extrapolation. Il en existe deux types: les arbres de classification qui permettent une prédiction de classe, et les arbres de régression qui permettent de prédire une quantité (valeur). Dans notre cas, nous sommes intéressés par les arbres de classification. Cependant, il faut porter une attention particulière à la profondeur maximale de l'arbre. Plus l'arbre sera profond, plus il sera précis (dans la limite d'overfitting), mais plus il sera coûteux en termes d'exécution et de mémoire. Plusieurs versions d'arbres de décision existent, nous choisissons

CART, algorithme implémenté dans scikit [137]. Nous garderons le nom de Decision Tree (DT) tout au long du manuscrit pour plus de lisibilité.

- **Les algorithmes bayésiens :** Ce sont des méthodes qui appliquent explicitement le théorème de Bayes pour des problèmes tels que la classification et la régression. Un modèle très connu dans cette catégorie est le Naive Bayes (NB) qui suppose que l'existence d'une caractéristique pour une classe est indépendante de l'existence d'autres caractéristiques. C'est donc un modèle probabiliste dont les données d'entrée ne sont pas corrélées lors de l'apprentissage pour trouver moyennes et variances des différentes variables. Ce type de classifieur, malgré les hypothèses simplistes, reste généralement performant. De plus, un avantage majeur est qu'il requiert relativement peu de données d'entraînement pour estimer les paramètres nécessaires à la classification, fonctionnalité très intéressante dans notre cas. Nous pensons donc que l'étudier est une option intéressante.

- **Les algorithmes de regroupement :** Les méthodes de regroupement sont souvent basées sur un apprentissage non supervisé. Elle vise à diviser un ensemble de données en différents *cluster* homogènes qui présentent des caractéristiques communes. Ce type d'algorithme est très utilisé pour distinguer 2 voix d'un enregistrement audio par exemple. Dans notre cas, il semble très peu intéressant, du fait que les comportements normaux et d'attaques ne sont pas très bien définis.

- **Les algorithmes d'apprentissage des règles d'association :** Les méthodes d'apprentissage des règles d'association extraient les règles qui expliquent le mieux les relations observées entre les variables dans les données. Ces méthodes sont généralement plutôt utilisées dans le *data mining* ou encore sur de grands ensembles de données pour créer des profils commerciaux typiques. Leur but est de trouver une corrélation entre plusieurs données dans de très grands amas de données. Ce n'est pas un modèle qui correspond à notre cahier des charges.

- **Les algorithmes de réseaux de neurones artificiels :** Les réseaux de neurones artificiels sont des modèles qui s'inspirent de la structure et/ou de la fonction des réseaux neuronaux biologiques. Ils sont généralement optimisés par des méthodes d'apprentissages probabilistes (souvent bayésiennes). Les réseaux de neurones ont un processus d'apprentissage généralement long, qui repose sur beaucoup d'exemples de cas réels. Cela convient très bien pour de la météorologie ou de la reconnaissance d'image par exemple. Cependant, dans notre cas, nous ne connaissons pas le comportement d'attaque, nous ne pouvons donc pas entraîner le modèle correctement. De plus, ce type de modèle est généralement lourd (temps d'exécution). Ainsi, cette famille n'est pas un bon candidat.

- **Les algorithmes d'apprentissage profond :** L'apprentissage profond est une version plus poussée des réseaux de neurones artificiels, avec de nombreuses couches cachées. Ils sont très efficaces et intéressants pour de l'analyse vidéo ou audio, très peu dans notre cas d'usage.

- **Les algorithmes de réduction de dimension :** Tout comme les méthodes de regroupement, la réduction de dimension cherche et exploite la structure inhérente aux données. Deux méthodes bien connues qui ont fait leurs preuves sont Quadratic Discriminant Analysis (QDA) et Linear Discriminant Analysis (LDA). Elles

sont intéressantes, car elles sont très légères à l'exécution, et sans paramètres à régler. Bien que les techniques de réduction de dimension soient majoritairement utilisées pour éliminer ou extraire des caractéristiques intéressantes, ils sont également utilisés pour classifier des données. Ces deux méthodes sont donc à prendre en compte, une avec un aspect plutôt linéaire et l'autre avec un aspect quadratique, qui la rend plus flexible.

- **Les algorithmes d'ensemble :** L'apprentissage à base d'ensemble s'appuie sur plusieurs algorithmes d'apprentissage pour obtenir de meilleures prédictions. Par exemple, les forêts d'arbres décisionnels (Random Forest (RF)), reposent sur des méthodes d'arbres de décision. La méthode implique la création de plusieurs arbres de décision en utilisant des ensembles de données créés à partir des données d'origine et en sélectionnant de manière aléatoire un sous-ensemble à chaque étape de l'arbre de décision. L'objectif est de réduire les erreurs possibles sur un arbre de décision classique. Cependant, la taille du modèle et son temps de prédiction s'en retrouvent fortement augmentés. Nous choisissons tout de même de l'évaluer, pour les mêmes raisons que les arbres de décision.

Cette préétude nous amène donc sur une liste de 8 algorithmes supervisés de machine learning (tableau 4.6). Si ces modèles sélectionnés sont capables de faire de la détection d'anomalies de manière supervisée, d'autres sont expressément créés pour et sont capables de le faire de manière non supervisée (voire semi-supervisée), on parle d'algorithmes de détection de valeurs aberrantes ou de nouveautés. La détection de valeurs aberrantes repose sur un apprentissage non supervisé alors que la détection de nouveautés repose sur un apprentissage semi-supervisé. Dans ce dernier cas, cela signifie que seules les valeurs correctes sont utilisées dans l'apprentissage (donc il n'est pas nécessaire d'avoir des comportements d'attaques). Dans le contexte de la détection des valeurs aberrantes, les anomalies ne peuvent pas former un groupe dense, car les estimateurs disponibles supposent que les valeurs aberrantes/anomalies sont situées dans des régions de faible densité. Au contraire, dans le contexte de la détection des nouveautés, les nouveautés/anomalies peuvent former un groupe dense tant qu'elles se trouvent dans une région de faible densité des données d'entraînement, considérée comme normale dans ce contexte. Nous proposons alors d'ajouter des modèles semi-supervisés de détection de nouveauté à nos tests :

- **Local Outlier Factor (LOF) :** Le modèle LOF est basé sur un concept de densité locale, où la localité est donnée par les k voisins les plus proches, dont la distance est utilisée pour estimer la densité. En comparant la densité locale d'un objet aux densités locales de ses voisins, on peut identifier des régions de densité similaire, et des points qui ont une densité sensiblement inférieure à celle de leurs voisins. Ces points sont considérés comme des valeurs aberrantes. Cette méthode s'approche du modèle KNN, et hérite ainsi des mêmes problèmes de performances et de coût mémoire des modèles (plus il y a d'échantillons, plus le modèle est lourd). Cependant, c'est un modèle très intéressant pour la détection d'anomalies, et qui a notamment fait ses preuves dans des applications de détection d'intrusions réseau. Il peut aussi bien s'appliquer dans une approche valeur aberrante ou nouveauté. Dans cette étude, nous le testons sous l'approche détection de nouveauté.

- **Isolation Forest (IF) :** IF est un algorithme d'apprentissage non supervisé pour la détection d'anomalies qui fonctionne sur le principe de l'isolation des anomalies. La construction se base sur des arbres isolés (iTrees), et les anomalies sont les points qui ont des longueurs moyennes de chemin plus courtes sur les iTrees. Ce modèle peut être entraîné seulement avec des données dites "normale" et ne demande pas une grande quantité de données à stocker pour son modèle. Dans ce cas, il s'agit d'un modèle semi-supervisé. Cependant, ce modèle a tendance à mieux fonctionner lorsque l'utilisateur fournit une contamination. La contamination est la proportion de valeurs aberrantes dans l'ensemble de données. C'est utilisé lors de l'ajustement pour définir le seuil de la fonction de décision. Cependant, dans notre cas nous ne donnons pas d'exemple de cas aberrants, nous avons donc une contamination nulle qui pourrait introduire des erreurs de classification. C'est un modèle qui reste très intéressant comparé aux autres modèles, car il a tendance à essayer d'isoler les anomalies au lieu du comportement normal. Ainsi, c'est un modèle qui va générer potentiellement plus de faux positifs, mais moins de faux négatifs. Nous l'incluons alors dans nos tests.

- **Elliptic Envelope (EE) :** Une façon courante de procéder à la détection des valeurs aberrantes est de supposer que les données régulières proviennent d'une distribution connue (par exemple gaussienne). L'idée ici est donc de créer un modèle en déterminant la "forme" de ces données et d'identifier comme anomalies tout ce qui sort de cette forme. Par exemple, toutes les valeurs dans une gaussienne sont considérées comme "bonne" alors que tout ce qui se trouve à l'extérieur est considéré comme une anomalie. C'est une approche qui peut être réductrice dans notre cas d'usage, mais nous proposons tout de même de le tester, car les coûts de stockage et d'exécution sont très faibles. La classification consiste en une simple analyse de covariance.

Nous avons décidé d'évaluer séparément les modèles supervisés de ceux de détection de nouveauté, car leur mode d'apprentissage est bien différent. Pour la méthode de détection, il est très intéressant d'utiliser un algorithme de détection de nouveauté même si ce dernier se révèle moins précis et plus lourd qu'un modèle supervisé, car il permet à l'utilisateur de ne pas faire d'apprentissage à partir du comportement d'attaque. Nous sélectionnons alors les 3 modèles de détection de nouveauté. Pour plus de clarté dans les résultats, nous proposons de réduire la liste des modèles supervisés à 4. Pour cela, nous les évaluons de manière empirique avec une approche exhaustive. Nous avons déjà testé tous les modèles avec différentes combinaisons d'évènements pour évaluer leur moyenne de précision. Ici nous proposons d'évaluer leurs performances, leurs poids et leur précision, basés sur la combinaison sélectionnée d'évènements. Cette évaluation, en plus des intuitions déjà évoquées, permettra de faire la sélection finale des modèles. Il est à noter que tous les modèles sont utilisés avec leurs paramètres par défauts. Ce choix peut être discutable, mais l'idée ici était de faire une sélection rapide des modèles candidats, de manière empirique.

Dans cette comparaison, nous avons réutilisé le jeu de données créé pour la sélection des HPC, mais nous avons gardé seulement les évènements sélectionnés. La figure 4.16 illustre la comparaison de la précision des modèles supervisés tandis que la figure 4.17 illustre la comparaison de la précision des modèles de détection de nouveautés

Tableau 4.6 – *Présélection de modèles de machine learning*

No.	Modèle de machine learning	Catégorie	Mode
1	Logistic Regression (LR)	Régression	Supervisé
2	Linear Discriminant Analysis (LDA)	Réduction de dimension	Supervisé
3	Support Vector Machine (SVM)	Instance	Supervisé
4	Quadratic Discriminant Analysis (QDA)	Réduction de dimension	Supervisé
5	Random Forest (RF)	Ensemble	Supervisé
6	K-Nearest Neighbors (KNN)	Instance	Supervisé
7	Naive Bayes (NB)	Bayésien	Supervisé
8	Decision Tree (DT)	Arbre de décision	Supervisé
9	Isolation Forest (IF)	Détection de nouveauté	Semi Supervisé
10	Local Outlier Factor (LOF)	Détection de nouveauté	Semi Supervisé
11	Elliptic Envelope (EE)	Détection de nouveauté	Semi Supervisé

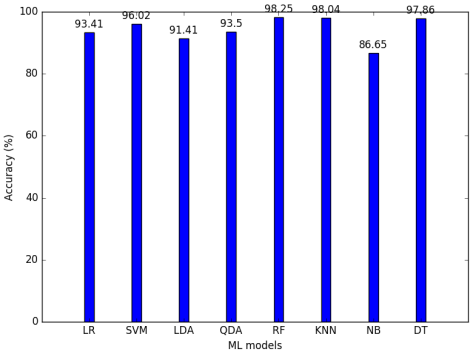

FIGURE 4.16 – *Comparaison de la précision des modèles de machine learning supervisés.*

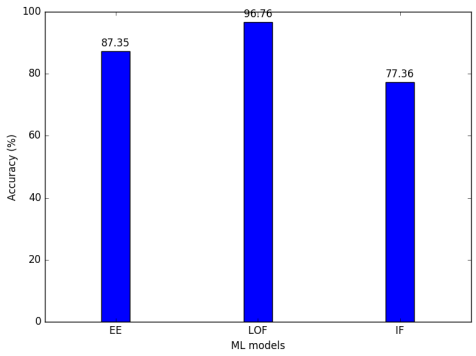

FIGURE 4.17 – *Comparaison de la précision des modèles de machine learning semi-supervisés.*

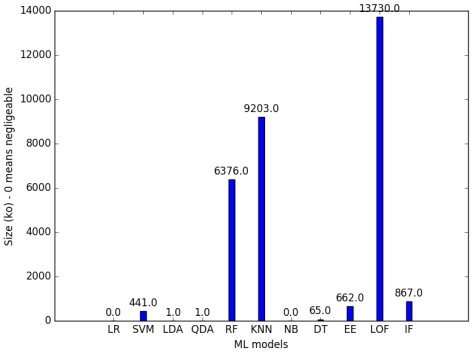

FIGURE 4.18 – *Comparaison de la taille des modèles de machine learning.*

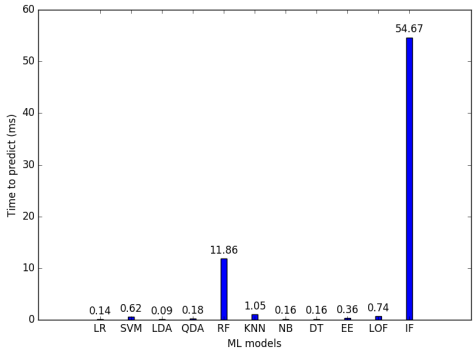

FIGURE 4.19 – *Comparaison de la rapidité des modèles de machine learning.*

(semi-supervisé). Dans un deuxième temps, nous avons comparé les performances de tous les modèles. La figure 4.19 montre le temps nécessaire pour prédire une sortie. La figure 4.18 indique l'espace requis sur le système pour stocker le modèle entraîné. Les données pour les modèles de détection de nouveauté sont données à titre indicatif, pour les comparer avec les modèles supervisés. Par exemple, le modèle LOF demande un espace de stockage excessif (13 730 088 octets) alors qu'il dispose d'un court délai pour prédire la sortie (0,74 ms) et d'une bonne précision (96,76%). Ce stockage conséquent est dû à la construction de la base de données nécessaire contenant les densités locales. En revanche, le modèle Isolation Forest (IF) demande un espace de stockage plus petit (867 090 octets), mais un temps pour classifier beaucoup plus long (57,67 ms). Dans ce cas, l'espace de stockage et le temps d'exécution dépendent de la profondeur des arbres de la forêt. Le modèle EE, comme prévu grâce à une simple analyse de covariance, est beaucoup plus léger que ses homologues (662ko et 0.36ms). Les précisions sont discutables, mais il est à noter que ces modèles sont testés avec leurs paramètres par défaut. Cependant, le modèle IF est moins précis que ces homologues, à cause de la contamination et de son utilisation en mode détection de nouveautés.

Tableau 4.7 – *Liste de modèle de machine learning sélectionnés pour le module de détection*

No.	**Modèle de machine Learning**	**Catégorie**	**Mode**
1	Quadratic Discriminant Analysis (QDA)	Réduction de dimension	Supervisé
2	Random Forest (RF)	Ensemble	Supervisé
3	K-Nearest Neighbors (KNN)	Instance	Supervisé
4	Decision Tree (DT)	Arbre de décision	Supervisé
5	Isolation Forest (IF)	Détection de nouveauté	Semi Supervisé
6	Local Outlier Factor (LOF)	Détection de nouveauté	Semi Supervisé
7	Elliptic Envelope (EE)	Détection de nouveauté	Semi Supervisé

Pour continuer la sélection, nous avons analysé les modèles supervisés. Nous avons présélectionné parfois deux modèles pour une catégorie, l'idée ici est d'en garder un seul par catégorie, et de supprimer les moins performants. Par exemple, le modèle Naive Bayes (NB) a la plus basse précision (86.65%). On peut expliquer ce résultat par le fonctionnement même de NB. Toutes les entrées du modèle sont traitées indépendamment, alors que dans notre cas ces données forment un tout. Cette non-corrélation entre variables, plus les hypothèses simplistes des modèles bayésiens rendent ce modèle peu efficace pour notre problème. Nous avons ensuite deux modèles de la famille des algorithmes basés sur les instances: KNN et SVM. Si KNN a une meilleure précision (98.04%) que SVM (96.02%), ce premier est aussi plus lourd. C'est un choix difficile ici, mais nous avons fait le choix de garder KNN, pour garder le modèle avec la plus grande précision. Un autre duel est QDA avec LDA. Même constat de ce côté, il faut choisir entre précision et performance. Néanmoins, les coûts en performance sont très faibles dans les deux cas, nous choisissons donc de garder le modèle avec la meilleure précision : QDA. Le modèle DT a d'excellents résultats : une précision de 97.86%, et des pénalités presque nulles. RF, qui est un ensemble d'arbres de décision et vise à amé-

liorer les erreurs de DT, a effectivement une meilleure précision (98.25%). Cependant, les performances sont dégradées avec un fort coût de stockage et d'exécution. Nous choisissons tout de même de garder les deux. LR, malgré ses faibles pénalités dues à un modèle très simpliste, n'a pas un bon score. Nous choisissons donc de l'enlever.

Au final, nous sélectionnons un ensemble homogène de modèles, avec certains plus lourds, mais plus précis, et d'autres, moins précis, mais plus légers (tableau 4.7). La prochaine étape est donc d'approfondir l'analyse de ces modèles avec la sélection d'évènements proposée dans la section précédente, et de les expérimenter dans les scénarios 1, 2 et 3.

4.4 Expérimentations et résultats

Dans cette section, nous évaluons la méthode mise en place, avec les trois scénarios présentés dans la section 4.3.1. Pour chaque scénario, nous évaluons la précision et les erreurs de classification sous différentes conditions de charge du système, *i.e.*, No Load (NL), Full Load (FL), Mix Load (ML). L'objectif est de valider l'approche générale de la solution: détecter une attaque par corruption de mémoire en utilisant des HPC et des modèles de ML. De plus, ces résultats doivent nous permettre de vérifier nos hypothèses sur les modèles de ML, et de sélectionner celui qui nous semble le plus pertinent sur un ratio précision/performance.

4.4.1 Scénario n°1

La première étude de cas est basée sur le jeu de données n°1. Ce jeu de données ne contient pas de comportement d'attaque ou de phase d'exploitation. Comme expliqué, le but de ce scénario est de voir s'il est possible de détecter une vulnérabilité (sans exploitation) grâce aux HPC et modèles de ML. Pour cela, ce jeu de données contient 20 000 échantillons provenant des compteurs lors de l'exécution du programme vulnérable et du profileur en mode *complet*. Nous avons ensuite entraîné les différents modèles de ML sélectionnés et nous avons étudié la précision de ces modèles et la matrice de confusion correspondante. Les détails du jeu de données, du scénario et des codes utilisés sont définis dans la section 4.2.1.b.

La précision d'un modèle est le premier critère pour analyser l'efficacité du mécanisme de détection. Les tableaux 4.9 et 4.8 représentent la précision de détection pour l'ensemble de données 1 pour les modèles de ML supervisés et semi-supervisés. Le tableau 4.9 montre que les modèles semi-supervisés, IF, LOF et EE présentent respectivement une précision de 53,89%, 54,81% et 52,51% sous des conditions de charge mixte (MixL). Alors que les modèles supervisés, DT, QDA, RF, KNN présentent respectivement une précision de 52,86%, 54,57% et 54,21% et 51,75%.

Comme on peut le constater, la précision globale est très faible. Une précision de 50% revient à proposer un algorithme qui classifie aléatoirement les échantillons. Le premier constat est de dire que la méthode ne fonctionne pas pour détecter seulement

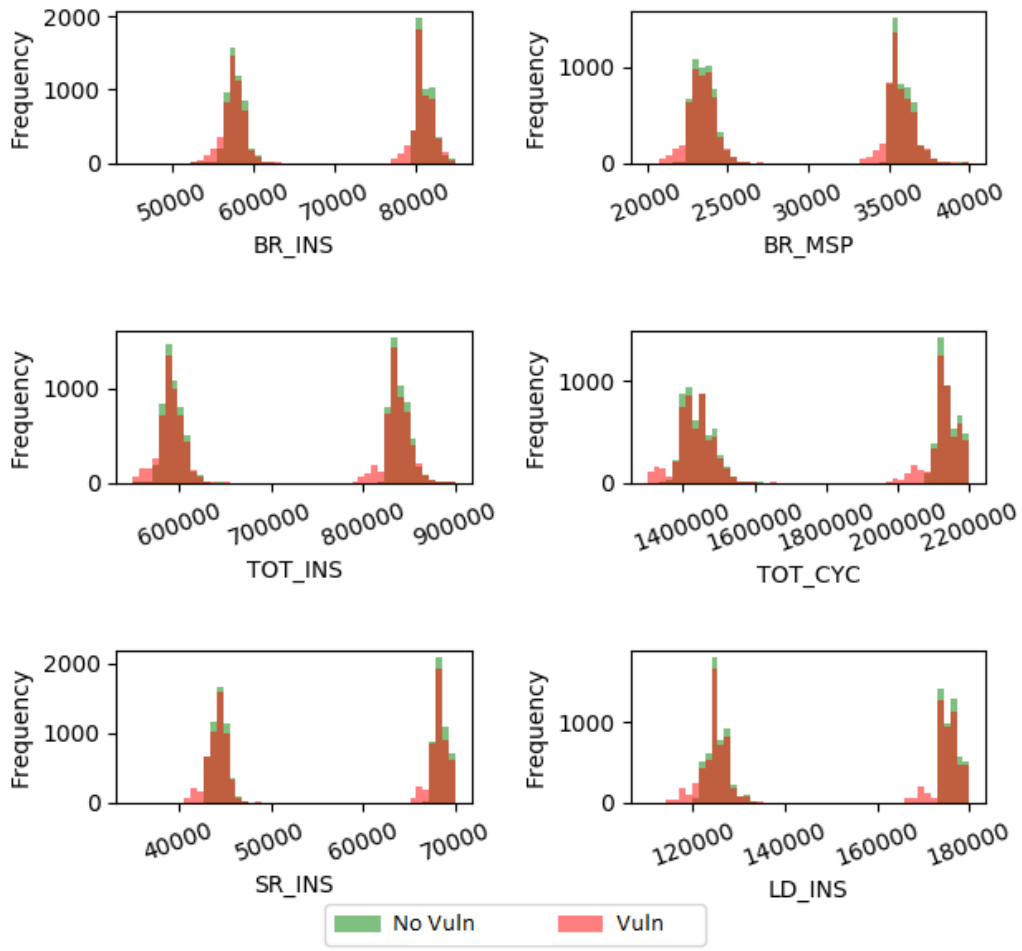

FIGURE 4.20 – *Distribution des valeurs des compteurs matériels sélectionnés pour le jeu de données n°1 sous des conditions de charge mixte (MixL).*

une vulnérabilité. En effet, l'apprentissage est fait grâce à des données comportementales du système venant des HPC, et dans ce cas d'étude nous cherchons à détecter une vulnérabilité dans le programme, sans phase d'exploitation. Le problème est que, une vulnérabilité, ne modifie pas forcément le comportement du programme, de sorte que les HPC ne fournissent pas des données exploitables. Par exemple, ici la vulnérabilité est l'écriture à un offset aléatoire d'un tampon, potentiellement hors limite (voir figure 4.21. La conséquence est le changement d'un octet dans la mémoire qui peut potentiellement provoquer un crash ou changer le flot de données. Il est complexe, voire impossible, de retrouver cette information à l'aide des compteurs, car il n'y a pas de changement de comportement assez fort pour être reflété sur les compteurs (hors crash). Cet argument devient évident en regardant la figure 4.20 qui illustre la distribution entre les données des compteurs venant de l'exécution d'un programme sans vulnérabilité et des données des compteurs venant de l'exécution du même programme avec une vulnérabilité (écriture tampon hors limite). On observe bien que ces deux comportements sont indiscernables et donc que l'entraînement d'un modèle de ML ne peut pas fournir de résultats exploitables. Cependant, même si ce cas d'étude ne fonctionne pas, il valide une première intuition: il n'est pas possible de discerner une vulnérabilité non exploitée dans un programme juste avec des évènements architecturaux/micro-architecturaux.

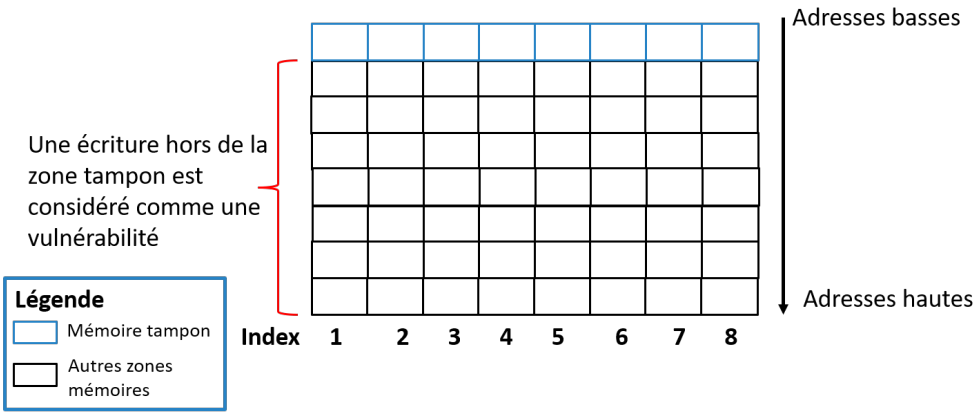

FIGURE 4.21 – *Exemple vulnérabilité d'écriture tampon hors limite.*

Tableau 4.8 – *Résultats de détection pour les modèles supervisés - Jeu de données 1.*

Modèle	Charge	Précision (%)	FP (%)	FN (%)
DT	**NL**	52.17	23.93	23.89
	FL	53.55	22.65	23.80
	MixL	52.86	23.30	23.85
QDA	**NL**	54.17	11.38	34.45
	FL	54.99	11.58	33.43
	MixL	54.57	11.48	33.95
RF	**NL**	53.09	21.39	25.53
	FL	55.35	20.09	24.56
	MixL	54.21	20.74	25.05
KNN	**NL**	51.50	23.57	24.93
	FL	52.01	23.02	24.97
	MixL	51.75	23.30	24.95

Tableau 4.9 – *Résultats de détection pour les modèles semi-supervisés - Jeu de données 1.*

Modèle	Charge	Précision (%)	FP (%)	FN (%)
IF	**NL**	53.25	14.27	32.48
	FL	54.55	8.75	36.70
	MixL	53.89	11.54	34.57
LOF	**NL**	54.87	0.84	44.30
	FL	54.75	0.79	44.46
	MixL	54.81	0.81	44.38
EE	**NL**	52.98	5.28	41.74
	FL	52.04	4.90	43.06
	MixL	52.51	5.09	42.40

4.4.2 Scénario n°2

La deuxième étude de cas proposée est basée sur l'ensemble de données n°2 comme défini dans la section 4.2.1.b. Ce scénario propose un jeu de données de 20 000 échantillons, avec une distribution égale entre les cas d'attaque et de non-attaque. Le profilage est fait en mode *complet*.

Si nous examinons la figure 4.22, tous les évènements matériels soigneusement choisis fournissent une distribution claire du comportement d'attaque par rapport au comportement de non-attaque. Il est évident que l'introduction de la phase d'exploitation dans les échantillons amène une différence nette entre l'exécution normale et anormale. De ce fait, les résultats de la précision, dans les tableaux 4.10 et 4.11, montrent que les résultats obtenus sont parfaits pour les modèles de ML supervisés et très bon pour les modèles de ML semi-supervisés. Les quatre modèles, DT, QDA, RF et KNN, présentent une précision de 100% dans les conditions de charge MixL. Ces quatre modèles ne se sont jamais trompés sur 20 000 échantillons.

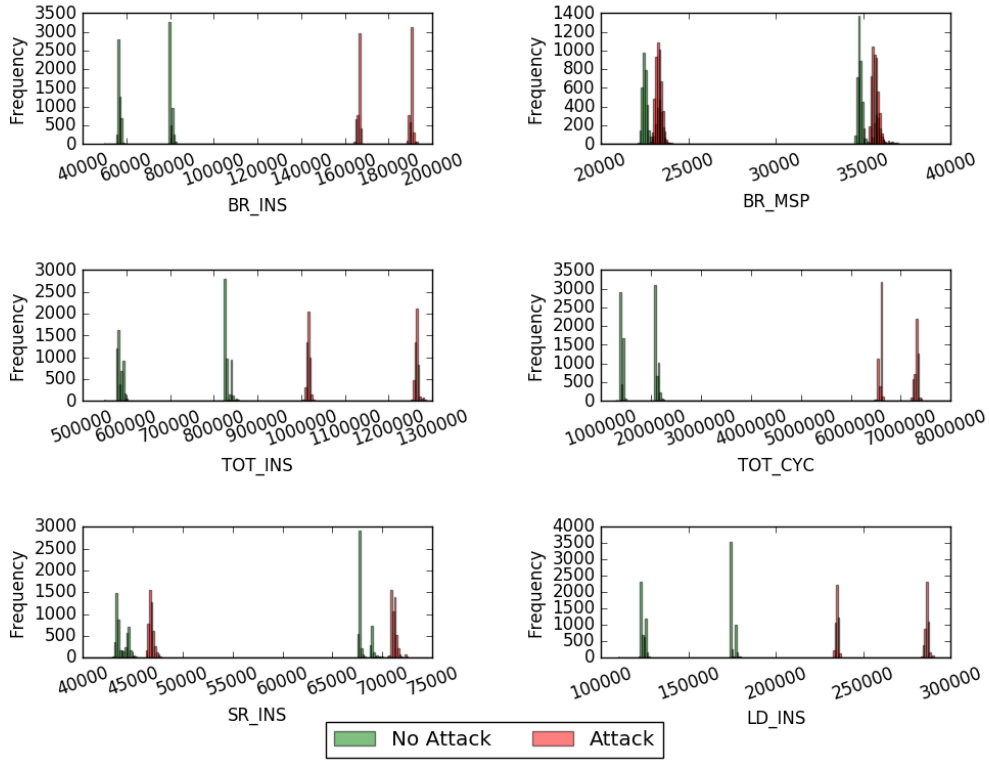

FIGURE 4.22 – *Distribution des valeurs des compteurs matériels sélectionnés pour le jeu de données n°2 sous des conditions de charge mixte (MixL).*

Le tableau 4.11 montre que les modèles semi-supervisés obtiennent également de bons résultats. Les modèles IF, LOF, et EE montrent une précision de 87,98%, 99,02%, et 94,83% dans les conditions de charge MixL. Ces modèles fonctionnent moins bien que les modèles supervisés dans ce cas. Une explication est que ces modèles tendent à détecter une valeur aberrante et si cette dernière n'est pas assez différente des valeurs "normales", alors l'échantillon est mal classifié. De plus, IF fonctionne moins bien que les autres modèles de sa catégorie, comme prévue, à cause de la contamination. LOF, configuré en mode détection de nouveautés, fonctionnent mieux que EE. En effet, nous avons expliqué que EE repose sur des modèles plus simplistes que LOF, mais ce dernier est bien plus lourd en termes d'exécution et de stockage mémoire.

Même si les résultats des modèles semi-supervisés (notamment ceux de LOF) sont encourageants, des explications s'imposent concernant les modèles supervisés et leur 100% de précision. Nous avons testé un modèle de type régression logistique (LR) sur ce jeu de données, et nous obtenons également 100% de précision. Un simple seuil, comme utilisé par ConFirm [127] ou Malone [128] pourrait classifier ce type de données de manière optimale également. Deux principales raisons :

1. D'une part, le mode *complet* du profilage permet cette facilité de classification. En effet, comme les valeurs des compteurs sont récoltées à la fin de l'exécution du programme, si une attaque a eu lieu pendant l'exécution, alors il est évident qu'elle aura impacté le nombre d'instructions exécutées par exemple. Et comme le mode *complet* permet une accumulation de la déviation plus grande sur les compteurs, il devient plus évident de classifier le problème.

2. D'autre part, le choix de notre programme et de l'attaque dans ce scénario impacte aussi la précision. La durée du programme en exécution normale est presque aussi longue

Tableau 4.10 – *Résultats de détection pour les modèles supervisés - Jeu de données 2.*

Modèle	Charge	Précision (%)	FP (%)	FN (%)
DT	**NL**	100	0.00	0.00
	FL	100	0.00	0.00
	MixL	100	0.00	0.00
QDA	**NL**	100	0.00	0.00
	FL	100	0.00	0.00
	MixL	100	0.00	0.00
RF	**NL**	100	0.00	0.00
	FL	100	0.00	0.00
	MixL	100	0.00	0.00
KNN	**NL**	100	0.00	0.00
	FL	100	0.00	0.00
	MixL	100	0.00	0.00

Tableau 4.11 – *Résultats de détection pour les modèles semi-supervisés - Jeu de données 2.*

Modèle	Charge	Précision (%)	FP (%)	FN (%)
IF	**NL**	86.03	13.97	0.00
	FL	89.90	10.10	0.00
	MixL	87.98	12.02	0.00
LOF	**NL**	99.10	0.90	0.00
	FL	98.95	1.05	0.00
	MixL	99.02	0.98	0.00
EE	**NL**	94.00	6.00	0.00
	FL	95.65	4.35	0.00
	MixL	94.83	5.17	0.00

que la durée de l'attaque. L'attaque fait donc fortement dévier le comportement du programme, et l'accumulation de la déviation se fait d'autant plus ressentir sur les compteurs. Les données deviennent alors très faciles à classifier.

Mais que se passerait-il si cette déviation était moins forte avec une autre attaque ou un programme plus long ? Si le programme était plus long, avec plus de complexité dans l'exécution du flot de contrôle, est-ce qu'il serait aussi trivial de classifier ces données ? Mais si le programme était plus long et plus complexe, est-ce qu'il reste pertinent d'utiliser le profilage en mode *complet* ? Il est finalement difficile de répondre à ces questions puisqu'il est impossible de généraliser le problème dans ces conditions, car la durée du programme à protéger et celle de l'attaque sont des inconnues dans l'équation.

Ce cas d'étude permet donc seulement de valider l'approche générale de notre solution, mais reste finalement peu intéressant en production. Profiler le programme en mode *complet* apporte trop d'inconnues dans l'équation, et une approche échantillonnée semble plus pertinente.

4.4.3 Scénario n°3

Bien que la détection de l'attaque une fois le programme terminé exige un faible surcoût car il est nécessaire de faire la classification qu'une seule fois par exécution complète, de nombreux arguments s'opposent à ce scénario :

1. Nous venons de voir qu'en mode *complet*, le problème n'est pas généralisable et que rien ne permet d'affirmer que la méthode de détection fonctionnera pour tout type de code et d'attaque.
2. Dans un système de détection, il est généralement nécessaire de détecter l'attaque le plus tôt possible, et non une fois que celle-ci est terminée.

Dans le cas d'étude n°3, nous expérimentons le mode *échantillonné* dans lequel l'attaque est détectée à différents moments de l'exécution, basé sur une période d'échantillonnage. Cette période d'échantillonnage permet de supprimer une inconnue du problème: la durée d'exécution du programme à protéger. Seule l'attaque reste inconnue. Nous définissons la granularité de la détection à 1ms pour une analyse grain fin et à 10ms pour grain épais. Cette granularité offre un compromis entre rapidité de détection et coût de la solution. Nous avons également réalisé l'expérience avec une granularité à 100us et 100ms. Cependant, nous ne présentons pas les résultats pour ces deux cas, car ils ne sont pas satisfaisants. L'échantillonnage à 100us impacte trop les performances, et celui à 100ms ne permet pas d'atteindre une précision intéressante.

Pour établir nos jeux de données en mode *échantillonné*, nous avons pris 1 million d'échantillons provenant des HPC lors de l'exécution d'un programme avec deux comportements distincts: normal et avec attaque (voir section 4.2.1.b). Pour rappel, notre jeu de données pour ce scénario contient plus d'échantillons du comportement normal que du comportement anormal. La distribution est 95% d'échantillons bons et 5% de mauvais. De cette façon, les modèles se concentrent sur l'apprentissage du bon comportement, et si à un moment donné, une activité anormale se produit, l'anomalie est capturée par les classifieurs ML entraînés.

Comme les jeux de données d'entraînement et de test contiennent une distribution inégale en matière de bon et mauvais cas, étudier seulement la précision des modèles peut ne pas être suffisant pour déterminer si les résultats sont corrects. Pour cette raison, les résultats de cette expérimentation sont présentés avec une évaluation détaillée sur la matrice de confusion (explication dans la section 4.2.1.c). De plus, nous proposons deux cas différents, utilisant deux jeux de données différents :

1. **Jeu de données similaires :** Un cas où l'attaque utilisée est la même lors de l'entraînement et lors du test.
2. **Jeu de données dissimilaires :** Un cas où il y a plusieurs attaques. Toutes les attaques lors de la phase de test sont différentes de l'entraînement.

Évidemment, aucune attaque n'est utilisée pour l'entraînement des modèles semi-supervisés.

4.4.3.a Détection sur des jeux de données similaires

La figure 4.23 montre la répartition des évènements matériels pour l'ensemble de données 3 sous des conditions de charge MixL avec un échantillonnage à grain fin. À la différence du jeu de données n°2 qui donne 100% de précision pour les modèles supervisés, ce type de cas est plus complexe. En effet, là où un modèle linéaire de type seuil aurait pu fonctionner pour le cas d'étude 2, on imagine clairement les limitations de ce dernier dans ce cas d'usage. Les données se superposent entre cas d'attaque et cas de non-attaque, et le machine learning prend alors tout son sens. En regardant les tableaux 4.12 et 4.13, nous pouvons observer les résultats de la détection à grains fins pour les modèles supervisés et semi-supervisés respectivement.

Le tableau 4.12 montre que la plupart des modèles de ML donne de bons résultats et sont peu enclins à une mauvaise classification. Tous les modèles donnent des résultats satisfaisants vis-à-vis des Vrais Positifs (TP) (supérieurs à 99%). Les modèles comme DT, RF ou encore KNN ont même des taux de Faux positifs (FP) inférieurs à 0,03%. Ces résultats sont très encourageants. L'analyse se corse un peu en étudiant les Faux Négatifs (FN). En effet, même s'ils restent relativement faibles, entre 0% et 2%, un FN en sécurité est une faille. C'est le modèle RF

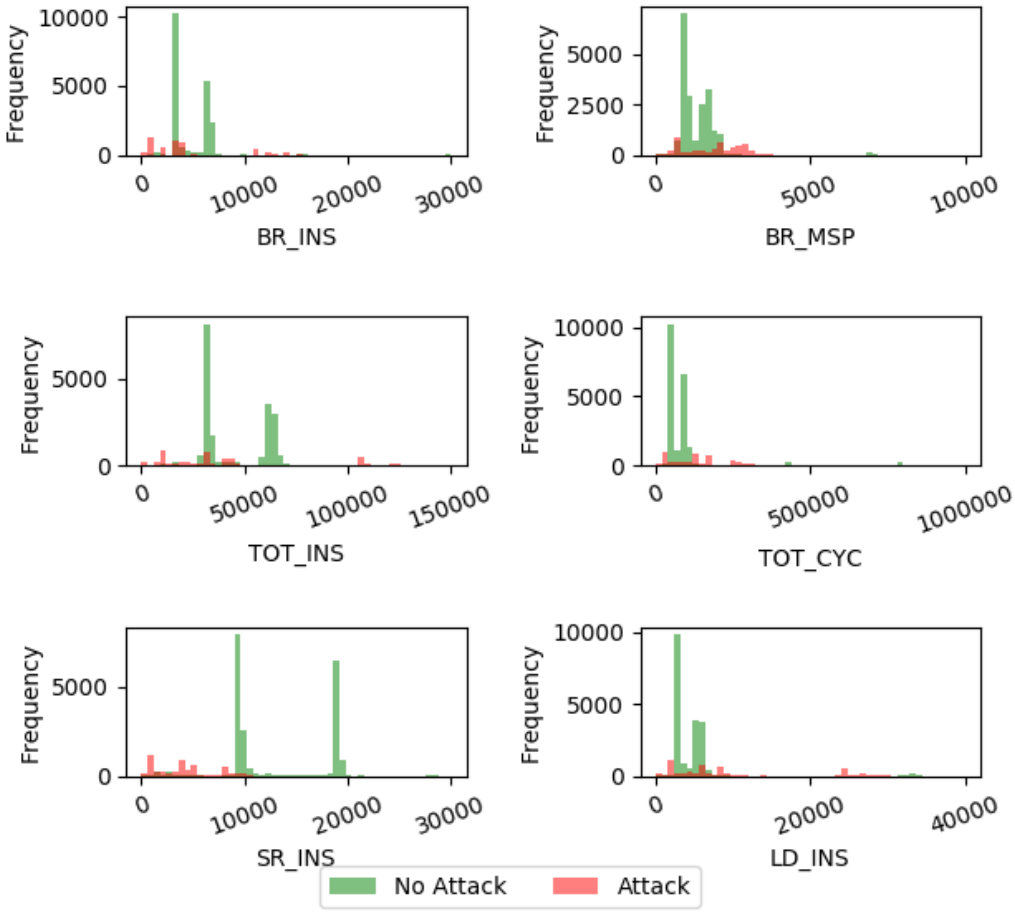

FIGURE 4.23 – *Distribution des valeurs des compteurs matériels sélectionnés pour le jeu de données n°3 sous des conditions de charge mixte (MixL) à un échantillonnage à grain fin.*

qui semble se tromper le moins souvent, aussi bien sur les FP que sur les FN quelles que soient les conditions de charge. Les meilleures performances du modèle RF par rapport au modèle DT sont évidentes. RF est basé sur plusieurs arbres de décision et est au final une amélioration, plus lourde, de DT. Le modèle KNN fonctionne moins bien que prévu, c'est dû à la faiblesse du modèle à construire une classification optimale lorsque la distribution de classe est asymétrique (95% vs 5% ici). Plus il y a d'exemples d'une classe dans le jeu de données plus le modèle tend à classifier un échantillon dans cette classe, car elle est statistiquement plus fréquente parmi les k plus proches voisins. Cela explique aussi son fort taux de TP. Même s'il existe des moyens de pondérer la classification pour contourner ces effets, nous n'avons pas réussi à obtenir un meilleur taux de faux négatifs.

Malgré ce taux global de FN, les résultats sont très intéressants pour les modèles supervisés. En examinant les modèles semi-supervisés dans le tableau 4.13, le premier constat est que les modèles LOF et EE ne fonctionnent pas comme espéré. En effet, leurs taux de faux négatifs est bien trop haut pour être utilisable, 31,99% et 87,89% sous des conditions de charge MixL. Plusieurs explications sont possibles:

- D'une part EE est un modèle qui définit une "forme" pour les données, et qui classifie ensuite bon/mauvais en fonction de cette forme. C'est une approche simpliste de détection de valeurs aberrantes. De ce fait, tous les échantillons mauvais produits par l'attaque ne sont pas considérés comme des valeurs aberrantes par le modèle (car peu de variance

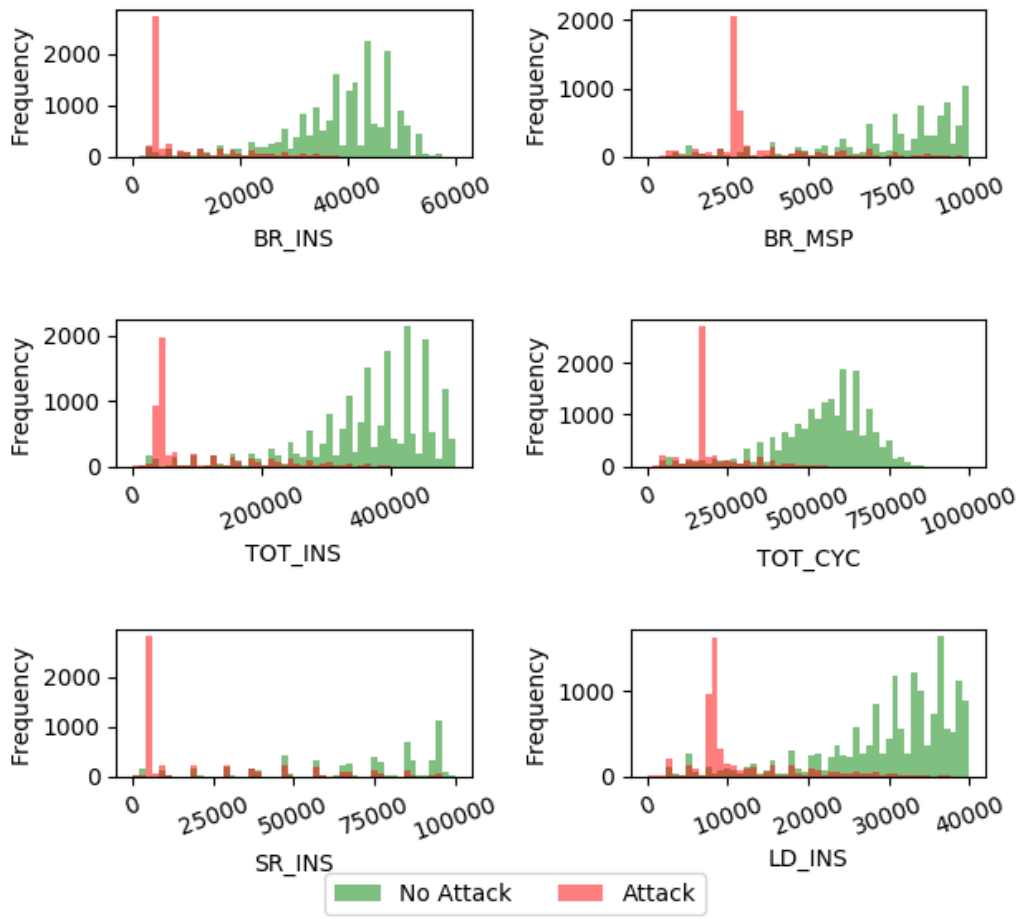

FIGURE 4.24 – *Distribution des valeurs des compteurs matériels sélectionnés pour le jeu de données n°3 sous des conditions de charge mixe (MixL) à un échantillonnage grain épais.*

avec des échantillons bons), ainsi ils sont mal classifiés.

- D'autre part, LOF est un modèle fonctionnant sur le même principe que KNN, grâce à la densité locale donnée par les k voisins les plus proches. Et de la même manière que pour EE, si des échantillons d'attaque se trouvent trop proches d'échantillons normaux, ils s'en retrouvent mal classifiés.

Le seul candidat intéressant dans ce cas présent est le modèle IF qui confirme nos intuitions. En effet, le fonctionnement du modèle diffère de LOF et EE, en essayant d'isoler les anomalies en priorité. De ce fait, le modèle ne produit aucun faux négatif, mais tend vers un taux de faux positifs conséquent. Ce type de comportement peut être acceptable dans une application où la sécurité est vraiment critique, et qui peut se permettre d'avoir des pénalités un peu plus élevées dues aux faux positifs.

Nous avons également testé la détection avec un profilage à grain épais, c'est-à-dire 10 ms. La figure 4.24 permet d'observer la distribution des échantillons pour ce scénario. Normalement, on suppose que la détection à grain épais devrait donner des résultats plus fiables que la détection à grain fin, car l'échantillonnage à grain épais permet d'analyser un échantillon plus long. De ce fait, les compteurs accumulent plus longtemps le comportement de l'attaque avant la classification (voir figure 4.25). Mais comme le montrent les tableaux 4.14 et 4.15, les résultats sont plus que mauvais. Que ce soient les modèles supervisés ou semi-supervisés, aucun ne fournit des résultats exploitables. La raison derrière cette baisse de performance est

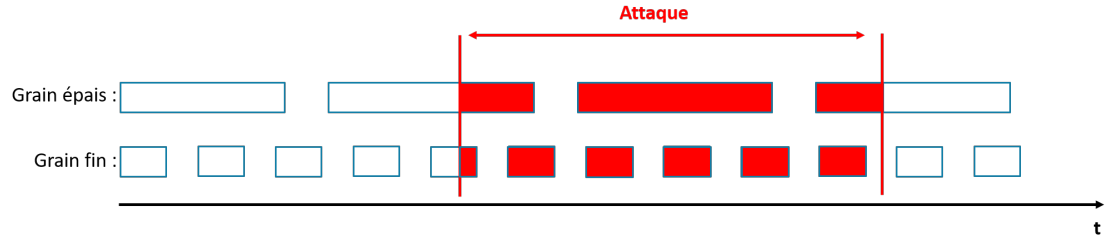

FIGURE 4.25 – *Comparaison échantillonnages à grain fin vs grain épais.*

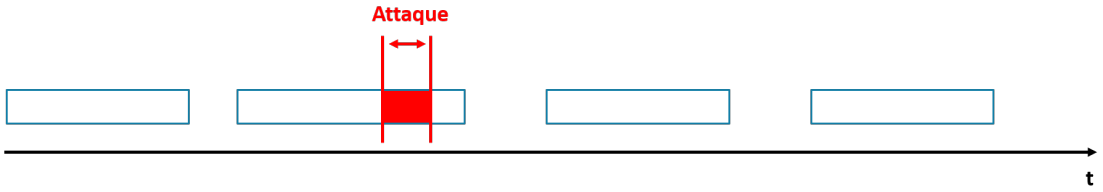

FIGURE 4.26 – *Génération d'un échantillon - Cas critique.*

que nos scénarios d'attaques utilisés sont très courts (de 1ms à 2ms). L'attaque termine donc son exécution dans un intervalle d'échantillonnage bien inférieur à 10 ms. Il en résulte qu'un échantillon est généré majoritairement pendant l'exécution du programme normal; 8ms à 9ms pendant l'exécution normale et 1 ms à 2ms pendant l'attaque (figure 4.26). Le comportement de l'attaque est donc noyé dans l'échantillon et devient difficile à classifier.

La précision des modèles dépend donc de la durée de l'attaque par rapport à la fréquence d'échantillonnage. C'est le même constat qui a été fait sur le scénario n°2, où la précision dépendait de la durée d'exécution du programme par rapport à la durée de l'attaque. Dans le cas d'un échantillonnage, minimiser la période permet donc de minimiser ces effets de bord, pour que l'échantillon soit généré au maximum pendant l'exécution de l'attaque. Cette période doit être fixée de manière à définir la durée minimale d'une attaque que l'ont veut détecter.

Tableau 4.12 – *Résultats de détection pour les modèles supervisés - Jeu de données 3 - Échantillonnage à grain fin.*

Modèle	Charge	TP (%)	TN (%)	FP (%)	FN (%)
DT	**NL**	99.97	99.01	0.03	0.99
	FL	99.99	99.30	0.01	0.70
	MixL	99.98	98.84	0.02	1.16
QDA	**NL**	99.71	98.02	0.29	1.98
	FL	99.03	100.0	0.97	0.00
	MixL	99.36	98.84	0.64	1.16
RF	**NL**	99.99	99.51	0.01	0.49
	FL	99.99	99.30	0.01	0.70
	MixL	99.99	99.42	0.01	0.58
KNN	**NL**	99.95	97.53	0.05	2.47
	FL	99.97	98.60	0.02	1.40
	MixL	99.97	98.26	0.03	1.74

Tableau 4.13 – *Résultats de détection pour les modèles semi-supervisés - Jeu de données 3 - Échantillonnage à grain fins.*

Modèle	Charge	TP (%)	TN (%)	FP (%)	FN (%)
IF	**NL**	88.55	100.00	11.45	0.00
	FL	89.91	100.00	10.09	0.00
	MixL	89.25	100.00	10.75	0.00
LOF	**NL**	95.40	63.33	4.60	36.61
	FL	95.92	73.48	4.08	26.52
	MixL	95.67	68.01	4.33	31.99
EE	**NL**	92.99	20.78	7.01	79.12
	FL	87.19	0.00	12.81	100.00
	MixL	90.01	12.11	9.99	87.89

Tableau 4.14 – *Résultats de détection pour les modèles supervisés - Jeu de données 3 - Échantillonnage à grain épais.*

Modèle	Charge	TP (%)	TN (%)	FP (%)	FN (%)
DT	**NL**	99.32	61.75	0.68	37.95
	FL	99.45	61.06	0.55	38.86
	MixL	99.39	61.48	0.61	38.43
QDA	**NL**	95.60	76.64	4.39	23.07
	FL	95.81	77.71	4.19	22.20
	MixL	95.71	76.85	4.29	23.15
RF	**NL**	99.98	58.78	0.02	41.67
	FL	99.99	58.68	0.01	41.24
	MixL	99.99	58.50	0.01	41.50
KNN	**NL**	99.91	58.78	0.09	40.92
	FL	99.94	58.68	0.06	41.24
	MixL	99.93	58.50	0.07	41.50

Tableau 4.15 – *Résultats de détection pour les modèles semi-supervisés - Jeu de données 3 - Échantillonnage à grain épais.*

Modèle	Charge	TP (%)	TN (%)	FP (%)	FN (%)
IF	**NL**	84.38	90.77	15.61	9.67
	FL	83.09	89.61	16.91	10.31
	MixL	83.72	89.91	16.28	9.99
LOF	**NL**	96.52	61.01	3.48	39.43
	FL	96.75	59.47	3.25	40.44
	MixL	96.64	59.94	3.36	39.96
EE	**NL**	89.41	61.01	10.59	38.69
	FL	90.93	57.89	9.07	42.03
	MixL	90.19	59.94	9.81	39.96

4.4.3.b Détection sur des jeux de données dissimilaires

Dans cette section, nous utilisons le même jeu de données d'entraînement que précédemment, mais nous changeons les attaques à détecter dans celui de test. L'idée derrière ce changement est de proposer aux modèles de ML d'essayer de détecter des scénarios d'attaques pour lesquels ils ne sont pas entraînés.

La répartition des évènements matériels liés à cette étude de cas peut être observée dans la figure 4.27. Les résultats présentés sont uniquement pour un échantillonnage à grain fin, le mode à grain épais étant écarté. Les tableaux 4.16 et 4.17 montrent les résultats de la matrice de confusion pour les modèles supervisés et semi-supervisés. On peut voir que le tableau 4.16 donne de bons résultats sur la décision de détection. Le taux de TP ne change pas vraiment entre ce scénario et le précédent, cependant on peut observer une petite diminution du taux de TN, environ 1% pour tous les modèles. Ces résultats sont clairement encourageants sur la capacité de ces modèles supervisés à détecter correctement des anomalies inconnues. Même si le taux de FN augmente en conséquence, il reste inférieur à 3% d'erreur sur tous les échantillons d'attaques. Le tableau 4.17 indique, sans surprise, que les modèles LOF et EE ne sont pas plus efficaces que dans le cas précédent. Par contre le modèle IF fonctionne toujours aussi bien avec aucune erreur de classification sur les comportements d'attaque et mais toujours un taux de faux positifs conséquent.

Pour conclure sur cette étude de cas, les modèles supervisés fonctionnent correctement pour détecter une anomalie, même dans le cas où ils n'ont pas été entraînés pour détecter une attaque spécifique. Il est bon de noter que dans la réalité c'est ce cas qui sera le plus fréquent. Un seul modèle semi-supervisé (IF) semble cependant être efficace, au détriment des faux positifs, et donc des performances.

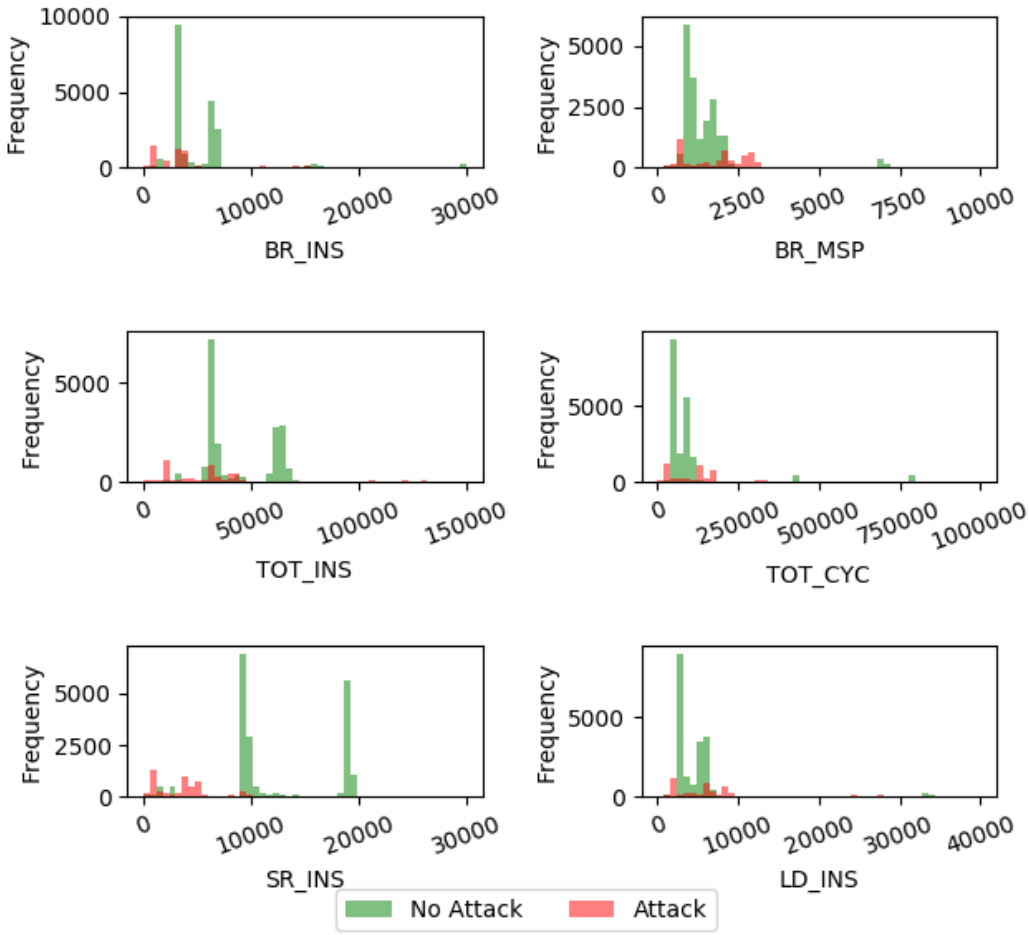

FIGURE 4.27 – *Distribution des valeurs des compteurs matériels sélectionnés pour le jeu de données n°3 sous des conditions de charge mixte (MixL) à un échantillonnage à grain fin - Jeu de données dissimilaire.*

Tableau 4.16 – *Résultats de détection pour les modèles supervisés - Jeu de données 3 - Échantillonnage à grain fin - Jeu de données dissimilaire.*

Modèle	Charge	TP (%)	TN (%)	FP (%)	FN (%)
DT	**NL**	99.97	97.60	0.03	2.40
	FL	99.94	98.30	0.06	1.70
	MixL	99.96	97.80	0.04	2.20
QDA	**NL**	99.67	98.02	0.33	2.14
	FL	98.39	100	1.61	0.00
	MixL	99.02	98.78	0.98	1.22
RF	**NL**	99.99	98.29	0.01	1.71
	FL	99.97	98.30	0.03	1.70
	MixL	99.98	98.29	0.02	1.71
KNN	**NL**	99.94	95.03	0.06	4.97
	FL	99.94	97.13	0.06	2.87
	MixL	99.94	95.84	0.06	4.16

Tableau 4.17 – *Résultats de détection pour les modèles semi-supervisés - Jeu de données 3 - Échantillonnage à grain fin - Jeu de données dissimilaire.*

Modèle	Charge	TP (%)	TN (%)	FP (%)	FN (%)
IF	**NL**	88.00	100.00	12.00	0.00
	FL	85.03	100.00	14.97	0.00
	MixL	86.49	100.00	13.51	0.00
LOF	**NL**	95.18	67.04	4.82	32.96
	FL	94.83	66.46	5.17	33.54
	MixL	95.00	66.99	5.00	33.01
EE	**NL**	92.74	14.98	7.26	85.02
	FL	91.73	0.00	8.27	100.00
	MixL	92.23	8.31	7.77	91.69

4.5 Coût de la solution

Les performances du mécanisme de détection proposé sont principalement classées en deux sous-catégories différentes : le mode *complet* et le mode *échantillonné*. Dans les sections suivantes, nous examinons la surcharge causée par les modèles de ML et le profilage. Les modèles de ML sont évalués selon des critères d'espace de stockage et de temps d'exécution (temps nécessaire pour classifier). Le profileur est évalué en fonction de la mémoire utilisée (RAM) et de l'occupation du CPU. Le mode *complet* est présenté pour un cas basique de 20 000 échantillons alors que le mode *échantillonné* est présenté pour 1 000 000 échantillons. Dans cette évaluation, nous présentons les résultats de coûts pour la configuration client/serveur du mécanisme. C'est-à-dire que le profileur est exécuté sur le système embarqué et que les chiffres présentés prennent en compte la communication des échantillons au PC. Les modèles de machine learning s'exécutent sur le PC.

4.5.1 Coût en mode *complet*

Comme expliqué dans la section 4.2.1, la décision de la classification en mode *complet* est prise après l'exécution complète du programme à protéger. Il n'y a donc pas de période d'échantillonnage impliquée dans la collecte des données des évènements matériels. Le coût de notre solution dépend principalement de 4 caractéristiques :

1. Le temps que les modèles de ML prennent pour prédire.
2. Le stockage mémoire (pour le modèle de ML).
3. L'utilisation du CPU (du système embarqué).
4. L'utilisation de la RAM (sur le système embarqué).

Comme le montre le tableau 4.18, pour 20 000 échantillons, la plupart des modèles ont une surcharge en matière de temps d'exécution relativement faible (<1ms). Deux modèles, RF et IF, sortent du lot avec des temps d'exécution de 8.84ms et 40.6ms. Ce coût d'exécution élevé était prévisible, car il faut traiter une multitude d'arbres de décision. Si on compose les forêts avec moins d'arbres, alors nous gagnerions en performance, mais nous perdrions en précision, pour tendre vers des modèles comme DT. Si l'utilisation de ces modèles reste envisageable en mode *complet*, nous allons voir que c'est problématique en mode *échantillonné*. Les modèles QDA et DT sont les plus rapides (0.11ms et 0.16ms) grâce à leur simplicité : QDA repose sur de l'analyse statistique et DT repose sur un simple arbre de décision. Ces détails permettent aussi d'avoir des modèles très légers en termes d'espace de stockage avec 1558 et 1561 octets respectivement. Ils montrent globalement moins de complexité pour être déployés.

Les modèles KNN et LOF sont similaires en termes de pénalités, prenant respectivement 0,61ms et 0,63ms pour prédire la décision et demandant 2 361 289 octets et 2 058 926 octets de mémoire de stockage. Cette similarité est due à leur fonctionnement, qui est basé sur la gestion des k plus proches voisins. Ce type de modèle est finalement peu intéressant d'un point de vue performance, car plus l'apprentissage sera conséquent, plus la taille du modèle généré augmentera, le rendant ainsi inutilisable sur un système embarqué.

Le tableau 4.19 montre le coût du profileur en mode *complet*, qui n'implique aucun échantillonnage. L'occupation du CPU peut être classée en deux catégories: *Système* et *Idle*. Le mode

Système signifie que le système a besoin de X% du CPU pour exécuter le profileur et le mode *Idle* signifie que le CPU est libre dans Y% du temps. Le tableau 4.19 montre que l'utilisation du CPU est <0,1% lorsque la collection de compteurs est en cours d'exécution et laisse ensuite plus de 99,99% d'utilisation du CPU pour les autres programmes. De plus, l'utilisation de mémoire RAM en mode *script* ne dépasse pas 1,2%, ce qui est négligeable.

Cette étude nous montre qu'en matière de surcharge du système, le profileur est négligeable en mode *complet*. De plus, certains modèles semblent sortir du lot. Pour les modèles supervisés, les modèles QDA et DT sont ceux qui demandent le moins de ressources. Les modèles semi-supervisés sont comme prévu plus gourmands, aussi bien en temps d'exécution qu'en espace de stockage. Nous avons déjà discuté cependant de leurs avantages indéniables.

Tableau 4.18 – *Coût des modèles de machine learning en mode* complet.

Modèle	Mode	Taille (Octets)	Temps d'exécution (ms)
DT	Supervisé	1561	0.11
QDA	Supervisé	1558	0.16
RF	Supervisé	64104	8.84
KNN	Supervisé	2361289	0.61
IF	Semi supervisé	933648	40.6
LOF	Semi supervisé	2058926	0.63
EE	Semi supervisé	101508	0.36

Tableau 4.19 – *Coût du profileur en mode* complet.

Taux d'échantillonnage (ms)	CPU Occupation	CPU utilisé (%)	Mémoire utilisée (%)
N/A	**Système**	< 0.1	1.2
	Idle	> 99.99	

4.5.2 Coût en mode *échantillonné*

En mode *échantillonné*, les résultats sont calculés pour 1 000 000 d'échantillons. Que ce soit pour le mode *complet* ou *échantillonné*, le temps d'exécution des modèles de ML ne varie que très peu: il ne dépend pas de la taille du jeu de données. Par exemple, les modèles comme KNN et LOF dépendent majoritairement du nombre de voisins recherchés. Plus ce nombre augmente, plus l'algorithme est lent. Les modèles comme RF, DT et IF dépendent de la profondeur des arbres de décision. Plus l'arbre est profond, plus l'algorithme est lent. La majeure différence due à la taille de nos jeux de données (20 000 vs 1 000 000 échantillons) est l'espace de stockage. C'est le cas pour les modèles DT, RF et IF, plus le jeu de données sera grand, plus il y aura de branches et d'arbres, augmentant ainsi la taille des modèles. Même constat du côté de KNN et LOF, plus il y a d'échantillons, plus la base de données représentant les voisins augmente, et donc la taille du modèle aussi. Le tableau 4.20 confirme ces analyses. Par exemple, les modèles DT et QDA s'exécutent en un temps similaire que précédemment. Le modèle DT prend 0,11ms pour prédire la décision, mais demande plus d'espace de stockage (56284 octets). Les modèles KNN et LOF prennent 120 746 869 octets et 205 621 014 octets d'espace de stockage, ce qui les rend inutilisables en production sur un système embarqué. De plus, les deux modèles IF et RF sont trop lents pour supporter notre taux d'échantillonnage à 1ms. En effet, le premier s'exécute en 41.63ms tandis que le second met plus de 9ms, bien supérieur à la période d'échantillonnage (1ms).

Dans le tableau 4.21, nous étudions le coût nécessaire à la mise en place du profileur en mode *échantillonné*. Avec un échantillonnage à grain fin de 1 ms, l'occupation du système et l'inactivité du CPU prennent 0,79% et 99,21% avec 1,2% d'utilisation de la mémoire (RAM). Comme nous pouvons l'observer, l'utilisation de la mémoire ne change pas entre les deux modes. Pour une période d'échantillonnage à 10ms, à grain épais, le profileur demande 0,30% d'activité CPU. La surcharge induite par le profileur reste très faible, voire négligeable, dans les deux modes. Nous avons augmenté encore plus la période d'échantillonnage, pour passer à 100us. Cependant cette rapidité implique beaucoup trop d'appels système et fait passer l'occupation du CPU à plus de 30% sans pour autant augmenter la précision de la détection.

Dans l'ensemble, le module de détection demande des ressources négligeables, que ce soit en mode *échantillonné* ou en mode *complet*. Les performances du profileur pour la collecte des compteurs matériels et l'envoi des données sont excellentes. Cette étude des performances, en plus de l'étude sur la précision et les matrices de confusion, permet d'établir la sélection d'un modèle optimal pour le module de détection.

Tableau 4.20 – *Coût des modèles de machine learning en mode* échantillonné.

Modèle	Mode	Taille (Octets)	Temps d'exécution (ms)
DT	Supervisé	56284	0.11
QDA	Supervisé	1558	0.19
RF	Supervisé	5849172	9.15
KNN	Supervisé	120746869	0.76
IF	Semi supervisé	579835	41.63
LOF	Semi supervisé	205621014	0.72
EE	Semi supervisé	9863464	0.39

Tableau 4.21 – *Coût du profileur en mode* échantillonné.

Taux d'échantillonnage (ms)	CPU Occupation	CPU utilisé (%)	Mémoire utilisée (%)
1	**Système**	0.79	1.2
	Idle	99.21	
10	**Système**	0.30	1.2
	Idle	99.70	

4.6 Généralisation

Suite aux expérimentations et à l'analyse de performances des modèles, nous proposons une évaluation finale pour sélectionner le meilleur candidat selon 3 critères à travers le tableau 4.22 :

- **Le taux de Vrai Négatifs (TN) :** il doit être supérieur à 98%, pour limiter au maximum les attaques non détectée. C'est le critère le plus important.
- **La taille des modèles :** elle doit être < 1Mo pour être envisageable sur un système embarqué.
- **Le temps d'exécution du modèle :** il doit être < 1ms pour respecter notre période d'échantillonnage minimale.

Tableau 4.22 – *Comparaison des modèles de machine learning.*

Modèle	Mode	TN > 98%	Taille < 1 Mo	Temps d'exécution < 1 ms
DT	Supervisé	Non	Oui	Oui
QDA	Supervisé	**Oui**	**Oui**	**Oui**
RF	Supervisé	Oui	Non	Non
KNN	Supervisé	Non	Non	Oui
IF	Semi supervisé	Oui	Oui	Non
LOF	Semi supervisé	Non	Non	Oui
EE	Semi supervisé	Non	Non	Oui

Le seul modèle respectant tous ces critères est le modèle QDA. Cependant, deux autres modèles pourraient être intéressants. Notre critère le plus important est le taux de TN, et les modèles RF et IF respectent ce taux. Ils sont cependant trop lourds pour la période d'échantillonnage définie. Mais, cette période peut être variable selon la granularité des attaques que l'on cherche à détecter, ainsi ces modèles pourraient devenir envisageables sous certaines conditions.

Nous proposons de continuer l'analyse du modèle QDA, en testant la méthode sur différentes applications de cryptographie, de tri, ou encore de recherche de chemins, sous des conditions de charge mixte. Nous avons sélectionné un total de 6 applications : AES256-CBC, MD5, SHA256, Base64, QuickSort et Dijkstra. Nous avons essayé de mixer des applications avec un fort impact sur les données et d'autres avec un fort impact sur le flot de contrôle, pour vérifier que la sélection des évènements soit efficace dans tous les cas. Pour chaque application, nous avons construit un jeu de données de 200 000 échantillons, avec une répartition inégale entre le comportement normal et les cas d'attaques (5% vs 95%), sous des conditions de charge mixte. Le profilage est fait en mode *échantillonné* toutes les 1ms. Le jeu d'entraînement contient un seul type d'attaque : une génération de shell. Les jeux de test contiennent différents type d'attaques : lecture fichier, écriture fichier, ouverture connexion réseau, suppression fichier, élévation de privilège. Le but est de tester que même si le modèle supervisé n'a pas appris à classifier ces attaques, il en reste capable.

Tableau 4.23 – *Résultats de détection pour le modèle QDA - Échantillonnage à grain fin.*

Programme	TP (%)	TN (%)	FP (%)	FN (%)
AES256-CBC	94.89	99.98	5.11	0.02
MD5	94.96	99.87	5.04	0.13
SHA256	96.59	99.95	3.41	0.05
Base64	94.83	99.88	5.17	0.12
QuickSort	99.54	97.79	0.46	2.21
Dijkstra	99.24	98.12	0.76	1.88

Les résultats sont présentés dans le tableau 4.23. Nous pouvons observer que les 4 applications de cryptographie et d'encodage ont un taux de TN très haut, avec très peu d'erreurs de classification sur les faux négatifs ($\tilde{<}$0.10%). En contrepartie, le taux de faux positifs est un peu plus élevé ($\tilde{5}$%) que les programmes de tri ou de recherche de chemins. Ces derniers ont un taux de FP supérieur à 99% avec très peu de FN (<2%). La sélection standard d'évènements semble donc bien fonctionner, avec une appétence pour les applications de cryptographie. Le modèle QDA est finalement intéressant dans ce cadre. De par sa nature visant la réduction de dimension, il a pour objectif d'identifier les caractéristiques les plus intéressantes pour un modèle.

De ce fait le modèle s'adapte à la sélection standard pour offrir les meilleures performances possibles, tout en impactant au minimum les performances.

4.7 Synthèse

Dans ce chapitre, nous avons étudié l'utilisation de modèles de ML et de HPC pour détecter des attaques par corruption de mémoire. L'objectif était de proposer un étage de détection qui soit léger et efficace, tout en gardant une précision de détection élevée. Nous avons détaillé l'approche et le raisonnement derrière cette technique, et proposé une implémentation de l'outil. Suite à cette implémentation, nous avons analysé la sélection des évènements appropriés et des modèles de machine learning, pour s'approcher du meilleur ratio pénalité/précision possible. Nous avons ensuite mis en place trois cas d'usages, en démontrant le fonctionnement de la solution pour les modes de profilage *complet* et *échantillonné*. Le choix final s'est porté sur le modèle supervisé QDA, que nous avons plus amplement testé de manière empirique sur différents algorithmes de cryptographie et de tri.

Toutes ces expérimentations ont amené plusieurs pistes de réflexion. Premièrement, en comparaison avec les techniques actuelles de détection d'anomalies, nous avons réussi à baisser drastiquement les coûts en performance. La solution de Malone *et al.* [128] augmente les performances de 10% et la solution de Tang *et al.* augmente de 40 à 100% les performances. Grâce à notre inférence réalisée sur un serveur et un profilage minimal, nous arrivons à descendre ces coûts de performance à moins de 1%. Il est complexe de se comparer à ces travaux en termes de précision, car les chiffres ne sont pas spécifiés, mais nous atteignons jusqu'à 98.70% de précision avec moins de 1% de faux négatifs et de faux positifs.

Une autre analyse que nous avons confirmée à partir d'expériences est que les solutions de détection simples basées sur un seuil ne sont pas assez intelligentes pour prédire une attaque sur des programmes complexes. Il est difficile de distinguer un comportement normal d'un comportement anormal, car d'une part, les attaques sont inconnues et peuvent se produire à n'importe quel moment, et d'autre part, les compteurs peuvent être bruités selon la charge sur le système. Par conséquent, distinguer les attaques en se contentant d'établir un seuil avec les informations provenant des HPC peut être une méthode naïve, entraînant beaucoup de mauvaises classifications.

4.7.1 Perspectives et limitations

Nombre de questions se posent suite à ces travaux. Nous avons défini deux modes de déploiement: le mode *autonome* et le mode *déporté*. Nous avons seulement présenté les résultats pour le deuxième mode, cependant il serait intéressant d'effectuer l'inférence du modèle sélectionné (QDA), pour un système embarqué, afin d'évaluer l'impact sur les performances. Cette inférence pourrait être faite en C pour déployer l'algorithme directement sur le processeur ou en VHDL pour déployer l'algorithme directement sur le FPGA dans le cas où le processeur serait hybride. De cette manière il serait d'une part possible de comparer le modèle en version logicielle et matérielle, puis de valider l'approche côté embarqué.

Une autre piste serait d'utiliser la temporalité entre les échantillons pour augmenter la précision des modèles. Dans les expérimentations que nous avons présentées, nous avons traité chaque échantillon indépendamment des autres. Cependant, dans le cas où l'attaque dure plus

longtemps qu'un seul échantillon, prendre en compte n échantillons lors de la classification peut être un facteur de réduction d'erreur. La temporalité du module de détection entre dans le cas de plusieurs pistes. Par exemple, la durée de l'attaque face à la période d'échantillonnage peut avoir des effets néfastes sur la précision de la détection. Nous proposons d'étudier ce comportement avec la figure 4.28. En effet, comme nous l'avons vu, il est possible qu'un échantillon ne contienne qu'une infime partie du comportement de l'attaque. Si l'attaque débute à t+3 et que la période d'échantillonnage dure pour 4 unités de temps, alors l'échantillon est correct de t à t+3 et mauvais de t+3 à t+4. Il contient donc 75% de comportement normal et à de fortes chances d'être classifié comme normal. Si l'attaque dure pour 6 unités de temps, alors, il y a de fortes chances que le prochain échantillon soit classé comme attaque (cas 1 dans la figure 4.28). Néanmoins, il est possible de mal classifier deux échantillons dans ce scénario. Dans le cas 2, la durée de l'attaque est trop courte pour impacter fortement un échantillon. Ici, la probabilité de détecter l'attaque reste donc très faible. Il est donc important de mieux étudier les effets de la période d'échantillonnage par rapport à la durée de l'attaque. Par exemple, serait-il pertinent d'augmenter la fréquence pour diminuer les coûts et garder un taux de précision élevé si l'on considère des attaques ou logiciels malveillants qui vont perdurer ?

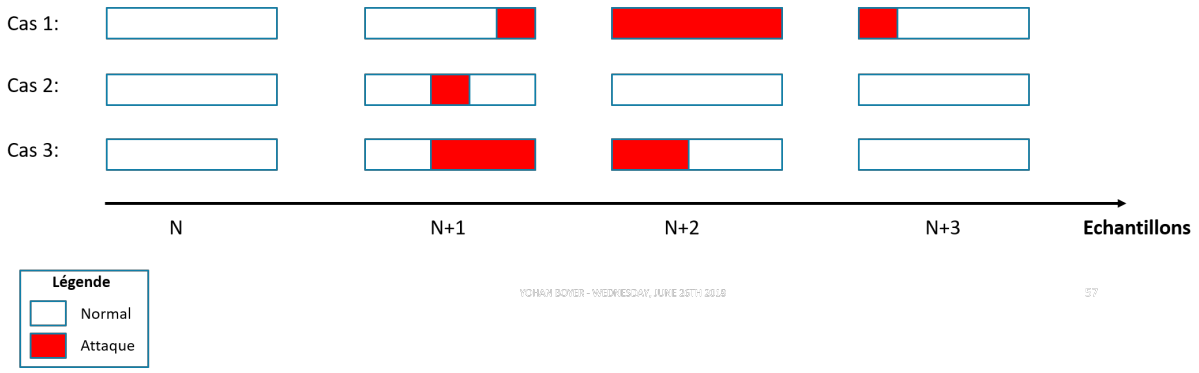

FIGURE 4.28 – *Contenu d'un échantillon vs classification.*

Une autre question complexe qui se profile et que nous avons déjà abordée est le problème de génération d'un jeu de données. En effet, il est nécessaire de labéliser chaque échantillon pour définir s'ils doivent être traités comme bon ou mauvais lors de l'entraînement. Cependant, faire un outil de génération de jeux de données automatique, et synchroniser parfaitement le moment de l'attaque, pose des questions. Si le début de l'attaque n'est pas parfaitement synchronisé avec le début de l'échantillon, alors on se retrouve dans le cas n°3 (figure 4.28), et il devient impossible de savoir si on doit labéliser l'élément comme bon ou comme mauvais. De plus, pendant l'exécution, rien ne certifie que l'attaque va être parfaitement synchronisée. Dans ce cas, est-il nécessaire de synchroniser parfaitement l'attaque et l'échantillonnage lors de l'apprentissage ? Si la synchronisation n'est pas faite, comment labéliser l'échantillon ? Est-ce que de cette manière, nous n'introduisons pas un biais dans le modèle, en labélisant comme mauvais un échantillon contenant potentiellement 95% du comportement normal du programme ? Nous n'avons pour le moment pas de réponses à ces questions, mais une étude plus poussée de ces concepts pourrait permettre d'augmenter la précision des modèles. Par exemple, si nous ne prenons pas en compte un échantillon contenant des deux comportements, alors il n'est plus possible de mal le classifier. En contrepartie, la latence de détection augmente.

Enfin, il se pose la question du déploiement de la contremesure suite à une alerte. En effet, nous avons vu que les modèles génèrent des erreurs de classification. Dans le cas de la sécurité, les faux négatifs sont à éviter à tout prix, car ils consistent en une faille de sécurité très grave; l'attaque est présente, mais non détectée. Les faux positifs sont moins graves, mais consistent en une surcharge du système inutile; le système déclenche la contremesure suite à une détection d'attaque qui n'en est pas une. Il est toujours possible d'affiner les modèles pour éviter au

maximum ces erreurs, cependant, nous voulons fournir un étage de détection facilement modulable, rapidement mis en place et qui ne nécessite pas d'intervention humaine lors de l'apprentissage. C'est pourquoi les modèles ne pourront vraisemblablement jamais être parfaits. Alors, il est important d'étudier les méthodes de rétablissement. Deux idées se confrontent:

- L'ajout d'un étage de diagnostic, visant à minimiser et éliminer les faux positifs après une alerte.
- L'étude d'une contremesure avec un coût minimal, n'impactant pas les performances du système, aussi bien lors du déploiement de la solution que de la remise en marche du service.

Nous proposons d'étudier ces deux approches lors du chapitre suivant.

V

Rétablissement du système

Sommaire

5.1 Motivations

Dans le chapitre précédent, nous avons proposé une méthode de détection d'anomalies à l'aide de machine learning, basée sur le profilage du comportement d'une application à partir des compteurs de performance. Nous avons démontré que la précision de la méthode peut monter jusqu'à 98.70% avec le modèle QDA, tout en générant très peu d'erreurs de classification. Cependant, si on prend en compte ces erreurs une fois une alerte levée, il peut être nécessaire de traiter l'information. Est-ce vraiment une attaque ? Est-ce un bug ? Est-ce un faux positif ? Dans le cas des faux positifs, il pourrait être intéressant d'étudier comment les filtrer. Dans le cas d'une attaque, nous avons vu que lorsqu'une alerte est levée, il est possible que le système soit déjà compromis. C'est dans ce cadre global qu'interviennent les méthodes de rétablissement du système illustrées par la figure 5.1 (en vert). Ces méthodes sont finalement aussi importantes que celles de détection, car si l'on se place dans le cas où l'on détecte 100% des attaques, mais que l'étage d'action n'est pas efficace, alors la détection ne sert à rien. Au contraire, si le taux de détection n'est pas assez élevé, alors, quelle que soit la contremesure, le système ne pourra pas se défendre. Nous proposons alors de détailler le traitement de l'alerte qui se décompose en trois étapes principales :

1. **Une étape de diagnostic** visant à déterminer si l'alerte de l'étage de détection correspond à une attaque, et si oui, laquelle. De cette manière, il est par exemple possible d'éliminer les faux positifs qui ne sont finalement que des fausses alertes.

2. **Une étape de décision**, qui, suite au diagnostic, doit choisir la contremesure à appliquer.

3. **Une action**, qui représente la contremesure à appliquer.

Dans ce chapitre, nous proposons d'étudier ces trois étapes pour évaluer la sécurité et la confiance à accorder aux méthodes existantes.

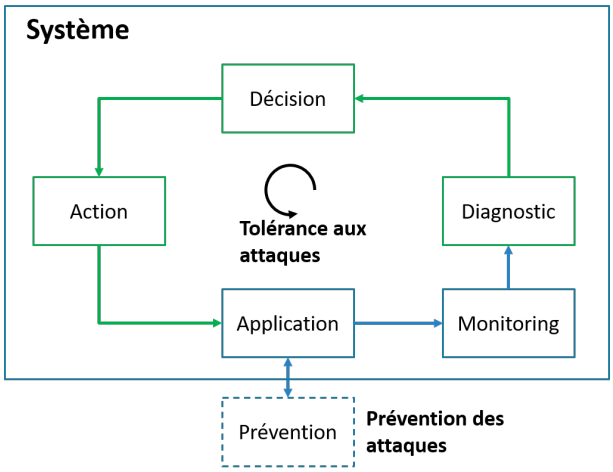

FIGURE 5.1 – *Boucle de sécurisation d'un système - Rétablissement ciblé.*

5.2 Diagnostic du système

5.2.1 Contexte

Un des problèmes majeurs identifiés lors de l'étude de la méthode de détection développée et testée au chapitre 4 est la gestion d'erreurs de classification. En effet notre solution utilise des algorithmes de machine learning pour traiter les données des HPC et qualifier l'état du système (comportement normal ou anormal). Cependant, comme nous l'avons vu, ces modèles impliquent aussi des résultats erronés comme des faux positifs et des faux négatifs. L'idée est donc de trouver un moyen pour réduire le taux d'erreur, non pas en affinant les modèles et les jeux de données, mais plutôt en proposant un outil de diagnostic pour essayer de traiter et éliminer ces erreurs à partir de l'exécution du programme. Ici, deux cas distincts se profilent:

1. Le premier est le cas des faux positifs. Une alerte est levée pour indiquer une anomalie sur le système alors que ce dernier est dans un état normal. C'est finalement une fausse alerte, qui lance le rétablissement du système à travers une contremesure, et pénalise fortement le système. Ici, le diagnostic peut être lancé après chaque alerte pour vérifier si elle est pertinente avant de lancer une contremesure. Ainsi, les faux positifs peuvent être éliminés.
2. Le deuxième cas correspond aux faux négatifs. Ici, aucune alerte n'est levée, mais l'attaque est en train de s'exécuter. Ce type de cas est plus compliqué à détecter. En effet, il n'y a pas d'alerte, il est donc impossible de lancer un diagnostic après cette dernière. C'est pourquoi il est important de construire des modèles ne produisant aucun faux négatif.

Les méthodes actuelles traitent toutes les alertes comme une attaque et déploient directement la contremesure. Dans ce chapitre, nous proposons d'étudier la faisabilité, les avantages et les limitations qu'apporterait un étage de diagnostic.

5.2.2 Hypothèses

Une erreur de classification en détection d'anomalies peut être associée à deux causes principales:

1. Les erreurs dues à une incapacité du modèle à classifier certains cas, car trop similaires.
2. Les erreurs dues a un apprentissage trop faible. C'est-à-dire, comme expliqué dans le chapitre 4, que la couverture du comportement normal du programme n'a pas été complète. Ainsi, à l'exécution, lorsqu'un cas non couvert par l'apprentissage se présente, il est traité comme une anomalie et produit une alerte.

Ici, le rôle du diagnostic serait de vérifier l'alerte, pour l'éliminer en cas de faux positif, et éventuellement la réinjecter dans l'apprentissage pour tendre vers un modèle plus complet. Pour mettre en place une telle méthode, nous proposons de diagnostiquer directement le programme qui s'exécute. Dans ce cas, nous cherchons à trouver un changement dans le flot de contrôle du programme. Pour répondre à ces besoins, il est nécessaire de respecter plusieurs critères :

1. Le diagnostic doit être déclenché à la demande. À la différence d'une méthode de détection qui s'exécute en tâche de fond pour lever une alerte, le diagnostic n'est lancé qu'après une alerte pour limiter l'impact sur les performances.

2. Le diagnostic traite les faux positifs. Ainsi, il est nécessaire qu'il soit fiable à 100% car s'il se trompe dans le traitement de l'alerte, nous introduirions une vulnérabilité critique dans le système.

3. La solution se doit d'être légère, pour être envisageable dans un système embarqué. C'est-à-dire qu'il faut minimiser au maximum les performances requises pour son exécution. De plus, le diagnostic ne doit pas être plus pénalisant que la contremesure. En effet, si ce premier est plus long que la contremesure, il est alors préférable de lancer directement cette dernière, sans traiter les faux positifs.

4. De la même manière que pour l'étage de détection, la solution doit facilement être adaptable pour protéger n'importe quel programme, sur tous les processeurs, pour n'importe quelle architecture.

À travers cette section, nous nous focalisons sur l'approche théorique, les concepts et les possibilités d'implémentation d'une telle méthode.

5.2.3 Méthodologie

Pour proposer un outil de diagnostic du flot de contrôle, nous nous inspirons du fonctionnement des CFI, étudiés dans le chapitre 3. Le diagnostic se doit d'être fiable à 100%, et les CFI permettent d'atteindre cette fiabilité sous certaines conditions, contrairement aux méthodes qui reposent sur des modèles comportementaux. Un CFI fonctionne en même temps que le programme qu'il protège, et vérifie durant l'exécution si le flot de contrôle du programme est correct. Pour cela, une méthode très connue est l'utilisation de graphe de flot de contrôle (CFG). Le CFG est établi hors ligne, basé sur l'étude, généralement statique, du binaire du programme. Lors de l'exécution, un flux d'information est dérivé pour les comparer avec celles du CFG. Elles peuvent prendre différentes formes, l'adresse d'une instruction, son *opcode*, un label, ses opérandes ou même le *hash* d'un bloc de base. Si une différence est repérée, alors une alerte est levée. Comme nous l'avons vu, cette méthode est très efficace pour détecter une attaque à l'exécution, mais souffre cependant soit de grosses pénalités en performance pour les versions logicielles, soit de problèmes de déploiement pour les solutions matérielles. Nous proposons d'essayer d'adapter la méthode, en prenant en compte les différentes limitations évoquées, pour diagnostiquer le flot de contrôle à la demande. L'idée finale étant de comparer le flot d'exécution d'un programme avec son CFG à un instant t.

La figure 5.2 introduit le concept de la méthode proposée. Comme les CFI, elle se décompose en trois parties distinctes :

- **Partie 1 (Construction d'un CFG):** De la même manière que pour un CFI, il est nécessaire de construire le graphe du flot de contrôle du programme à l'aide d'analyse statique. Le CFG va contenir tous les transferts possibles entre les blocs de base. Ce graphe est ensuite mis de côté pour une utilisation future et permettra de vérifier les branchements exécutés. Une question se pose alors: comment construire ce graphe de manière automatisée et complète ?

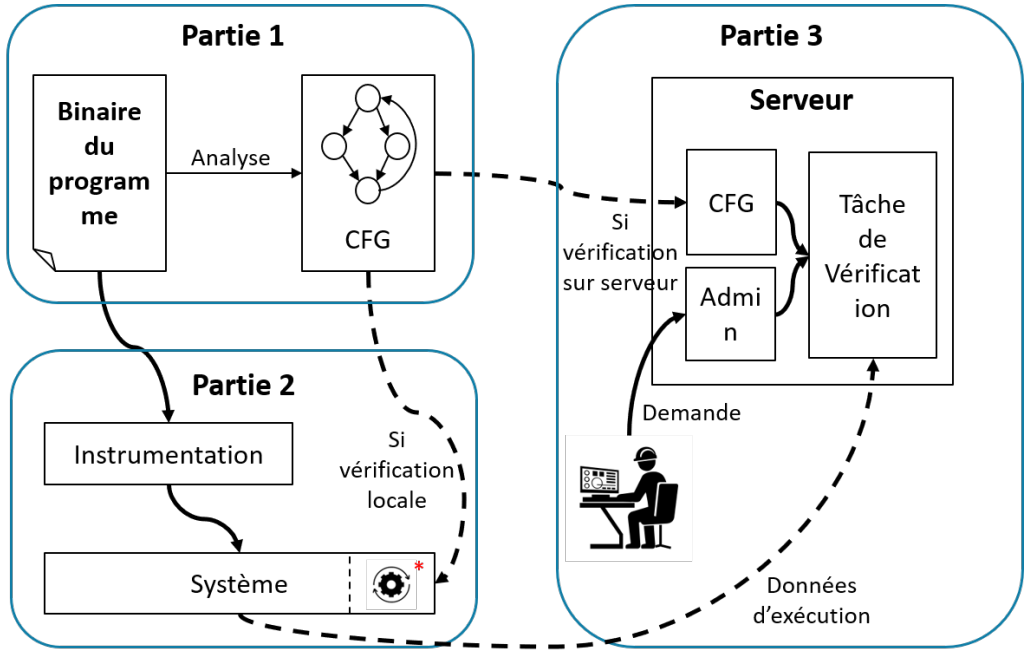

FIGURE 5.2 – *Méthodologie globale de l'outil de diagnostic.*

- **Partie 2 (Instrumentation):** Une fois le CFG construit, des données de l'exécution sont requises pour être comparées. La différence notable avec un CFI dans notre cas et qu'il faut garder ces informations jusqu'à ce qu'un diagnostic soit lancé. La comparaison n'est pas faite en temps réel. De nombreuses questions sont associées à cette partie. Quelles informations sélectionner ? Comment récupérer ces informations de manière à impacter le moins possible les performances du système ? Comment les stocker sur le système sans qu'elles puissent être altérées par l'attaquant ? Est-il envisageable de garder les données de l'exécution complète, ou faut-il plutôt en garder un simple échantillon ?

- **Partie 3 (Vérification):** le but final du diagnostic est de faire la vérification du flot de contrôle. Lors d'une approche CFI classique, le CFG du programme est d'abord construit hors ligne. Puis, des données sont récoltées lors de l'exécution du programme sur le système et sont directement comparées au CFG pour déterminer si l'exécution est autorisée. La vérification est donc relativement simple, car il suffit de comparer les graphes en temps réel. Dans le cas de notre outil de diagnostic, la vérification peut être lancée à n'importe quel moment et nous n'avons pas forcément les informations de l'exécution complète du programme. Donc, nous ne connaissons pas la temporalité entre le CFG et le graphe d'exécution et le point de départ de la recherche n'est pas connu. Comment faire pour retrouver le point de départ dans le CFG ? Est-il possible d'effectuer cette recherche depuis un serveur au lieu du système lui-même pour éviter trop d'impact sur les performances ? Est-ce qu'il existe des outils permettant cette recherche ? Est-ce que la complexité du code de recherche reste raisonnable ?

Le concept même de la méthode pose donc beaucoup de questions, et nous allons essayer de répondre à un maximum d'entre elles lors de ce chapitre. L'idée est d'étudier ces parties une à une, indépendamment, pour voir si la solution est possible, et dans le cas contraire, quelle(s) partie(s) serai(en)t bloquante(s).

5.2.3.a Construction du graphe de flot de contrôle

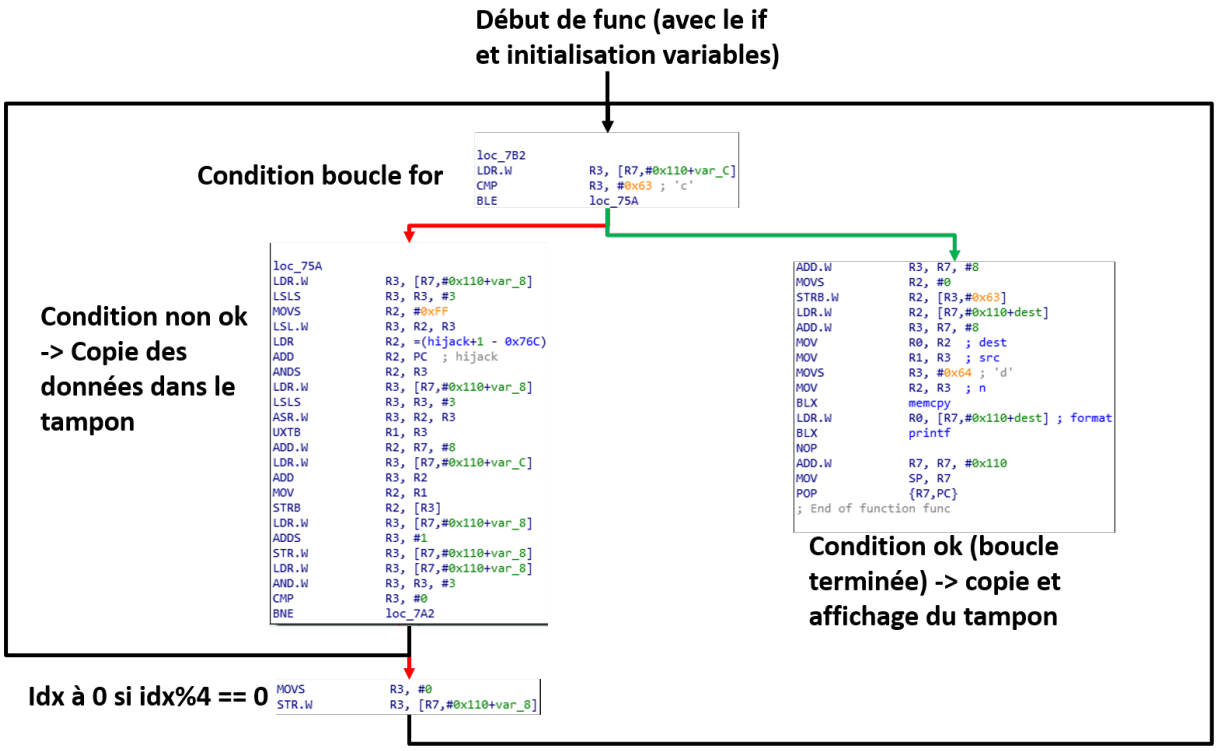

FIGURE 5.3 – *Exemple d'une partie du CFG du programme en annexe A*

Que ce soit dans le cadre de notre outil, dans la mise en place de CFI, ou tout simplement dans l'analyse de binaire, la construction d'un CFG est un prérequis. Cette technique est largement discutée dans la littérature et a fait l'objet de nombreuses publications [75, 138, 139, 140, 141, 142, 143]. Un CFG est un graphe composé de blocs de base (BB), représentés par les sommets, et d'instructions de changement de flot, représentées par les arrêtes. La construction d'un CFG à partir d'un binaire consiste à appliquer un algorithme récursif qui désassemble et analyse les blocs de base, puis identifie les changements de flot possibles. Le but est de connecter les blocs de base petit à petit à leurs successeurs. Nous pouvons observer sur la figure 5.3 le CFG de la fonction *func* du programme 2 en annexe (A). On peut voir que chaque condition fait l'objet d'un bloc de base, plus ou moins conséquent.

Si cette construction semble facile, en pratique, l'implémentation d'un outil est très complexe en raison de l'architecture cible. En effet, des langages tels que le C ont la particularité d'être multiplateformes. C'est-à-dire qu'ils vont aussi bien pouvoir être compilés pour une architecture ARM que pour du x86 ou encore des cibles AVR. Faire un outil de construction de graphe qui respecte toutes ces architectures demande dans un premier temps de passer le binaire dans une forme intermédiaire de représentation (IR) à travers des méthodes de *binary lifting*. Il existe plusieurs librairies permettant de créer cette représentation. Nous listons et comparons ces librairies dans le tableau 5.1 à l'aide de plusieurs critères:

1. La forme intermédiaire de représentation (IR).
2. S'il est fait spécifiquement pour de l'analyse statique.
3. Son prix; gratuit ou payant.
4. La facilité d'utilisation. Par exemple une API bien documentée.

5. Si il est disponible pour plusieurs architectures (ARM, x86, avr, etc).

Tableau 5.1 – *Comparaison des outils permettant de générer des graphes de flot de contrôle.*

Projet	**IR**	**Analyse statique**	**Prix**	**Facilité d'utilisation**	**Arch**
ANGR [75]	libVEX	Oui	Gratuit	Oui	Multi
BAP [76]	Custom (BIL)	Oui	Gratuit	Oui	Multi
AVRORA [144]	Custom	Oui	Gratuit	Oui	AVR
CLANG [145]	LLVM	Objectif compilation	Gratuit	Oui	Multi
IDA [146]	Custom	Oui	Payant	Oui	Multi
Ghidra [147]	Custom	Oui	Gratuit	Moyen	Multi
Radare2 [148]	ESIL	Oui	Gratuit	Non	Multi
X	REIL [149]	Oui	X	X	X

Une des IR les plus connues est libVEX [150], utilisée par Valgrind [71] et angr [75]. On retrouve d'autres IR comme REIL (*Reverse Engineering Intermediate Language*) [149], ESIL (*Universal IL*) [151], utilisé par radare2 [148], ou encore LLVM (*Low Level Virtual Machine*) [145], utilisé par CLANG. D'autres outils très connus utilisent leur propre IR comme IDA [146], Ghidra [147], BAP [76] avec BIL *(BAP Intermediate Language)*, ou encore avrora [144]. En théorie REIL est idéalement construit pour faire de l'analyse de binaire. Cependant il n'existe pas aujourd'hui d'implémentation complète de la solution. Le projet initial BinNavi a été abandonné après le rachat de l'entreprise par Google. Il existe aujourd'hui OpenREIL [152] qui implémente les concepts, mais reste basé sur des librairies comme libVEX ou BIL. De plus, il est actuellement non recommandé d'utiliser la librairie par ses développeurs, par manque de stabilité. LLVM est aussi une approche intéressante d'IR avec une large communauté active, mais est plus axé sur la compilation que sur l'analyse de binaire. L'IR de radare2, ESIL, est aussi réputé pour être efficace, mais est plus compliqué à prendre en main que les autres librairies. IDA possède toutes les qualités requises pour de l'analyse statique de binaires, mais demande une licence payante. Ghidra est un projet récent, qui manque encore de stabilité et de facilité d'utilisation. Cette étude nous amène donc sur ces deux candidats qui respectent tous nos critères : angr avec libVEX et BAP avec son propre IR (BIL).

Nous proposons alors l'utilisation de angr et présentons un exemple à partir du programme A. La figure 5.4 montre avec quelle simplicité il est possible de charger un programme et de construire son CFG. Un *warning* est affiché lors du chargement du binaire, car il est compilé en PIE, et n'a donc pas d'adresse de base pour sa section *.text*. Angr en fixe donc une par défaut (*0x40000000*). Une fois le CFG calculé, nous pouvons facilement accéder à tous les blocs de base du programme (voir figure 5.6).

Si nous prenons le premier bloc de base présenté dans la figure 5.3 (correspond au BB en *0x4007b3*), il est possible de récupérer toutes les informations nécessaires pour la mise en place de notre outil une fois le CFG chargé dans angr (voir figure 5.5). Par exemple, il y a des attributs

```
yohboy@ub18:~$ workon angr
(angr) yohboy@ub18:~$ python
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
>>> p = angr.Project('sample.out', load_options={'auto_load_libs': False})
WARNING | 2020-10-05 17:46:10,792 | cle.loader | The main binary is a position-independent executable. It is being loaded with a base add
ress of 0x400000.
>>> cfg = p.analyses.CFGEmulated(keep_state=True)
```

FIGURE 5.4 – *Chargement d'un programme dans angr.*

pour récupérer l'adresse d'un bloc de base, l'adresse de la fonction à laquelle appartient le BB, les adresses des instructions contenues dans un BB, ou encore la liste de ses successeurs. Par exemple, ici les deux successeurs sont les BB en *0x40075b* et *0x4007bb*, qui représentent respectivement les blocs *condition non ok* et *condition ok* de la figure 5.3. Angr fournit donc un large choix d'informations à utiliser pour faire la vérification finale entre le CFG et les données récupérées pendant l'exécution.

FIGURE 5.5 – *Exemple d'utilisation des blocs de base.*

Pour conclure, la construction de graphe de flot de contrôle est un sujet complexe, mais largement étudié par la communauté. Nous avons vu qu'il existe plusieurs outils, certains payants, d'autres gratuits, basés sur des approches différentes. Angr présente toutes les fonctionnalités nécessaires, mais aussi une simplicité d'utilisation appréciable. Cette étude répond donc à la majorité de nos questions concernant la partie 1.

```
>>> nodes = cfg.graph.nodes()
>>> for node in nodes:
...     print(node)
...
<CFGENode _start 0x40058d[52]>
<CFGENode 0x40052c[12]>
<CFGENode __libc_start_main 0x1000014>
<CFGENode __libc_csu_init 0x400821[20]>
<CFGENode _init 0x4004c4[8]>
<CFGENode call_weak_fn 0x4005d4[24]>
<CFGENode $a 0x4004cc[4]>
<CFGENode call_weak_fn+0x18 0x4005ec[4]>
<CFGENode __libc_csu_init+0x14 0x400835[8]>
<CFGENode call_weak_fn-0x9c 0x400538[12]>
<CFGENode __libc_csu_init+0x1c 0x40083d[16]>
<CFGENode __libc_csu_init+0x30 0x400851[4]>
<CFGENode PathTerminator 0x4d0>
<CFGENode frame_dummy 0x400699[2]>
<CFGENode __libc_start_main 0x100007c>
<CFGENode register_tm_clones 0x400625[24]>
<CFGENode main 0x4007f1[16]>
<CFGENode register_tm_clones+0x20 0x400645[2]>
<CFGENode register_tm_clones+0x18 0x40063d[8]>
<CFGENode main+0x24 0x400815[10]>
<CFGENode main+0x10 0x400801[12]>
<CFGENode __libc_csu_init+0x2c 0x40084d[4]>
<CFGENode __libc_start_main 0x1000080>
<CFGENode 0x400574[12]>
<CFGENode __libc_csu_init+0x1e 0x40083f[14]>
<CFGENode PathTerminator 0x1000080>
<CFGENode atoi 0x1000028>
<CFGENode __do_global_dtors_aux 0x400659[30]>
<CFGENode main+0x1c 0x40080d[8]>
<CFGENode __do_global_dtors_aux+0x1e 0x400677[4]>
<CFGENode __do_global_dtors_aux+0x2a 0x400683[2]>
<CFGENode $a 0x4004e4[12]>
<CFGENode func 0x400721[18]>
<CFGENode deregister_tm_clones 0x4005f9[16]>
<CFGENode PathTerminator 0x4d0>
<CFGENode func+0x1c 0x40073d[30]>
<CFGENode func+0x12 0x400733[10]>
<CFGENode deregister_tm_clones+0x18 0x400611[2]>
<CFGENode deregister_tm_clones+0x10 0x400609[8]>
<CFGENode func+0x92 0x4007b3[8]>
<CFGENode func+0x24 0x400745[22]>
<CFGENode __do_global_dtors_aux+0x22 0x40067b[10]>
<CFGENode func+0x3a 0x40075b[66]>
<CFGENode func+0x9a 0x4007bb[30]>
<CFGENode func+0x82 0x4007a3[6]>
<CFGENode func+0x7c 0x40079d[12]>
<CFGENode 0x400520[12]>
<CFGENode 0x40055c[12]>
<CFGENode 0x40055c[12]>
<CFGENode memcpy 0x1000010>
<CFGENode putchar 0x1000020>
<CFGENode putchar 0x1000020>
<CFGENode func+0xb8 0x4007d9[8]>
<CFGENode func+0x88 0x4007a9[18]>
<CFGENode 0x4004f0[12]>
<CFGENode PathTerminator 0x1000000>
<CFGENode func+0xc0 0x4007e1[10]>
<CFGENode PathTerminator 0x4004cc>
<CFGENode PathTerminator 0x4005c1>
<CFGENode PathTerminator 0x400677>
<CFGENode PathTerminator 0x4007e1>
```

FIGURE 5.6 – *Liste des blocs de base du programme en annexe A.*

5.2.3.b Instrumentation

La seconde étape consiste à récupérer les données nécessaires pendant l'exécution pour comparer le flot de contrôle dynamique avec celui du CFG. Les choix concernant les informations à récupérer peuvent être multiples. Nous proposons d'étudier une méthode qui se veut très simple: comparer l'enchaînement des blocs de base avec leurs adresses d'entrée et de sortie, *i.e.,* comparer l'adresse des instructions de changement de flot (*call,ret,jump*) entre celles prévues dans le CFG et celles réellement exécutées. Deux étapes sont nécessaires: la récupération de l'adresse de l'instruction exécutée et le stockage de cette dernière. Le stockage se doit d'être sécurisé, c'est-à-dire que la zone mémoire utilisée pour sauvegarder les adresses de l'enchaînement des blocs de base ne doit pas être accessible depuis le reste du code (et plus particulièrement par l'attaquant). Il faut donc penser un mécanisme pour isoler cette zone mémoire. De plus, le stockage du système n'est pas infini. Bien que nous pourrions imaginer que les données soient envoyées au fur et à mesure sur un serveur, cette solution prendrait trop de bande passante et demanderait trop de performance. L'idée est de garder seulement les N dernières instructions, N dépendant de la granularité de la recherche. Plus N est grand, plus

l'échantillon du chemin parcouru est grand, et plus le diagnostic est fiable. En contrepartie, la solution demande plus de mémoire, et la recherche dans le graphe est d'autant plus longue.

Les questions les plus complexes sont comment récupérer les instructions, et comment les stocker de manière sécurisée ? Pour les récupérer, nous avons exploré différentes solutions :

1. La modification matérielle du processeur (pipeline) pour fournir directement l'adresse de l'instruction exécutée.
2. L'instrumentation logicielle. Ici encore, plusieurs solutions possibles :
   - Utiliser une approche émulation, comme Valgrind [71], QEMU [153], ou encore Unicorn [154] qui est basé sur QEMU. Ainsi, il devient relativement facile d'ajouter un traitement supplémentaire à chaque instruction.
   - Utiliser une approche compilation, où le compilateur va directement insérer le code nécessaire à la récupération des adresses.
3. Des solutions mixtes :
   - Une modification du pipeline pour ajouter une instruction et une modification du compilateur pour ajouter cette instruction.
   - Une modification de l'émulateur pour ajouter une instruction et de la même manière, une modification du compilateur pour la prendre en compte.

La modification matérielle, bien qu'intéressante et probablement celle impactant le moins les performances du système, ne nous intéresse pas directement. Les raisons sont les mêmes que pour les CFI matériels. Il faut que la solution puisse être déployée sur tous les processeurs, même ceux ne bénéficiant pas de la modification matérielle. Nous optons alors pour une approche logicielle ou mixte.

Le cas de l'émulation

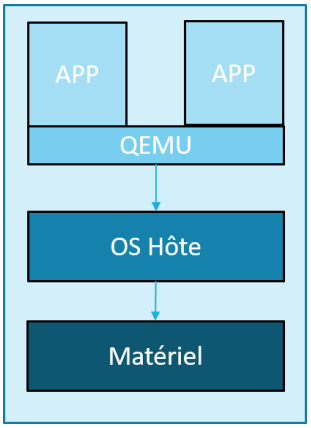

FIGURE 5.7 – *Émulation avec QEMU - Architecture.*

En informatique, l'émulation consiste à imiter un comportement physique par du logiciel. Un émulateur permet de simuler des architectures différentes sur un processeur. Par exemple, exécuter un programme ARM sur un processeur x86. Si on prend l'exemple de QEMU (figure

5.7), il fournit l'abstraction nécessaire pour exécuter des applications d'une autre architecture. Comme la description de cette dernière est faite de manière logicielle, il est possible de la modifier facilement. Ainsi nous pouvons imaginer un cas d'usage.

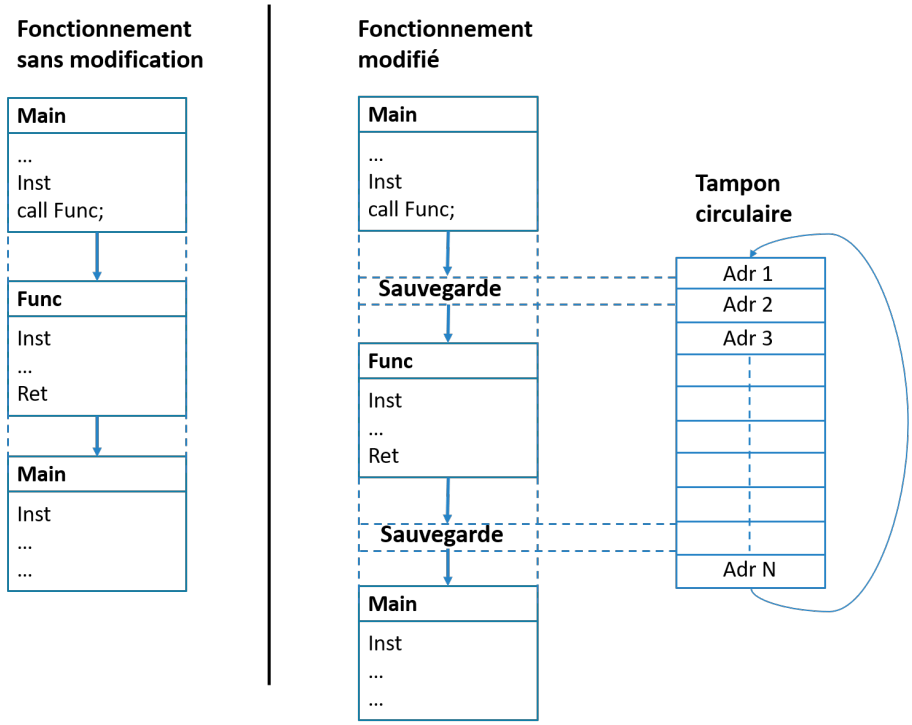

FIGURE 5.8 – *Modification de l'émulateur.*

Comme il est possible de modifier le comportement d'une instruction directement dans l'émulateur, l'idée est de venir modifier toutes les instructions de changement de flot pour qu'elles sauvegardent l'adresse de l'instruction exécutée avant de remplir leur rôle. Nous donnons un exemple dans la figure 5.8. Dans son fonctionnement normal, l'instruction *call* va simplement appeler la fonction spécifiée par son opérande (ici, *Func*). Dans la version modifiée pour notre outil, avant d'appeler *Func*, l'émulateur va d'abord sauvegarder l'adresse de l'instruction *call* pour garder une trace de son passage. Pour ça, nous proposons un tampon circulaire, avec une profondeur de N index. L'adresse est insérée dans le tableau, et l'exécution de la fonction peut continuer.

Évaluer la sécurité consiste d'une part à analyser les accès au tampon circulaire. Comme expliqué, ce tampon ne doit pas être accessible par le programme lui-même, sinon l'attaquant pourrait le falsifier. Puisque l'émulateur gère le fonctionnement de l'instruction et le remplissage du tampon circulaire, facile d'isoler la mémoire nécessaire au tampon. Ainsi, l'attaquant n'a donc aucun contrôle sur cette zone mémoire. Cependant, l'utilisation d'un émulateur dans ce cadre peut être problématique. En effet, l'exécution du programme peut être ralentie par des ratios de 500% à 1000%. Donc si la solution reste intéressante pour construire des prototypes et tester la méthode, elle semble peu envisageable dans le cadre d'un déploiement sur système embarqué.

Le cas de la compilation

L'instrumentation basée sur la compilation consiste à ajouter dans le code compilé les éléments nécessaires pour une nouvelle fonctionnalité. Dans notre cas, cela pourrait être l'ajout

d'instructions existantes permettant d'écrire dans un tampon avant chaque instruction de changement de flot. Au lieu de modifier le comportement d'une instruction, du code est ajouté directement dans le binaire. Cependant, la méthode présente un avantage considérable par rapport à l'émulation: il y a moins d'impact sur les performances. Lors de la compilation du programme, on peut imaginer réserver un registre du processeur pour contenir l'index du tampon circulaire. Avant chaque instruction de changement de flot, la valeur du *program counter* (PC) est stockée à l'aide d'une instruction *mov* et l'index est incrémenté de 1 si inférieur à N sinon il est réinitialisé à 0. Le fonctionnement est très simple et les pénalités sont donc très inférieures à celles proposées par l'émulation. Cependant, il reste un problème de taille à régler : les accès sécurisés au tampon. En effet, comme c'est le programme qui s'occupe de gérer le tampon circulaire, alors un attaquant pourrait lui aussi y accéder pour écraser les valeurs. Une autre question d'intégrité se pose. Le code nécessaire pour remplir le tampon circulaire est directement ajouté avant l'instruction de branchement. En cas d'attaque par réutilisation de code, l'attaquant pourrait simplement appeler les instructions de changement de flot, en évitant le remplissage du tampon, et ainsi contourner complètement la solution. Cet aspect est en fait limité, car le remplissage est effectué juste avant l'instruction de changement de flot. Or, pour mener une attaque par réutilisation de code, l'attaquant a besoin d'autres instructions que celles de branchement, donc il sera dans l'obligation de passer par le remplissage du tampon. L'attaquant ne peut pas contourner la solution.

Au final, l'instrumentation du programme à la compilation, bien que peu coûteuse, ne permet pas de définir une solution complètement sécurisée, car elle ne permet pas d'isoler la zone mémoire du tampon circulaire. La solution peut se révéler inefficace en cas d'attaque, donc non utilisable dans notre cas d'usage. Elle pourrait seulement servir d'outil d'analyse à distance en cas de bug du système.

Solution mixte

Une approche mixte pour ajouter une instruction dans le jeu d'instruction (basée sur la modification du pipeline ou de l'émulateur) et l'insérer dans le code compilé avant chaque *call, ret, jump* permettrait de sécuriser l'accès au tampon. En effet, on peut très bien imaginer que seule cette instruction puisse accéder au tampon circulaire, ainsi l'attaquant ne pourrait ni lire ni écrire dedans. Cette solution dans le cas de l'émulateur reste non envisageable en raison des performances. Dans le cas de la modification matérielle, en comptant sur une modification du pipeline, la solution pourrait être idéale. Elle aurait très peu d'impact sur les performances grâce au remplissage du tampon circulaire en une instruction spécifique. Cependant, les anciens processeurs ne bénéficiant pas de la modification ne pourraient pas mettre en place le diagnostic.

Alternatives

Un compromis entre émulation et compilation pourrait être de définir un processus en charge d'exécuter le programme à diagnostiquer. Ce processus jouerait le rôle de l'émulation, sans pour autant émuler un processeur complet. L'idée est simplement de gérer le programme à protéger (de la même manière qu'un débogueur ou un traceur par exemple) pour qu'à chaque instruction de changement de flot, le processus remplisse le tampon circulaire. De ce fait, l'accès au tampon est sécurisé, car seul le processus en a connaissance. Si une telle solution reste envisageable, il faut évaluer les performances requises, et les possibilités d'implémentation pour avoir un processus permettant de tracer les instructions de changement de flot.

Une autre alternative est d'utiliser les *Program Trace Macrocell* [155] (PTM) de ARM par exemple. Un PTM est un module matériel de traçage en temps réel qui permet de suivre les instructions d'un processeur avec très peu d'impact sur les performances (<5%). L'idée est donc d'utiliser un processus qui se charge de tracer le programme avec le PTM pour remplir le tampon circulaire. De cette manière, l'impact de performance est fortement réduit. Cependant, le PTM ne fournit pas exactement l'adresse du branchement pris. Il indique uniquement si le branchement est pris ou non. Cela peut poser des problèmes, car comme nous gardons seulement les N derniers branchements parcourus, nous ne pouvons pas nous situer dans le CFG, et nous perdons le contexte d'exécution. Il faudrait embarquer un peu d'intelligence dans le processus traceur pour déterminer à partir des informations du PTM, le branchement exécuté. Cependant, on retombe dans des les mêmes mécanismes de CFI matériels dépendant d'une sonde de débog. Si la solution demande autant de performance qu'un CFI matériel, il est préférable d'implémenter directement ce dernier.

Finalement, nous avons étudié plusieurs solutions, mais aucune ne semble envisageable en l'état. Les méthodes basées sur un émulateur ont trop d'impacts sur les performances et ne sont pas viables dans le cadre d'un système embarqué. Celles basées sur l'instrumentation du compilateur uniquement n'offrent pas un niveau de sécurité suffisant. Les solutions mixtes basées sur l'ajout d'une instruction et d'instrumentation du compilateur sont intéressantes, mais limitent le déploiement de la méthode. Seules les solutions alternatives semblent encourageantes, offrant un niveau de sécurité suffisant, mais une analyse plus poussée est nécessaire pour s'assurer de pénalités en performances faibles.

5.2.3.c Vérification

L'étape de vérification consiste à vérifier que le parcours des blocs de base à l'exécution est bien identique à celui prévu dans le CFG. D'une part, nous avons vu comment construire le CFG, et d'autre part, nous avons vu différentes méthodes d'instrumentation pour récupérer les adresses des blocs de base parcourus. Même si cette dernière partie n'est pas complètement fonctionnelle, nous pouvons tout de même étudier la partie vérification, pour déterminer si la solution est envisageable, aussi bien en termes de faisabilité que de performance.

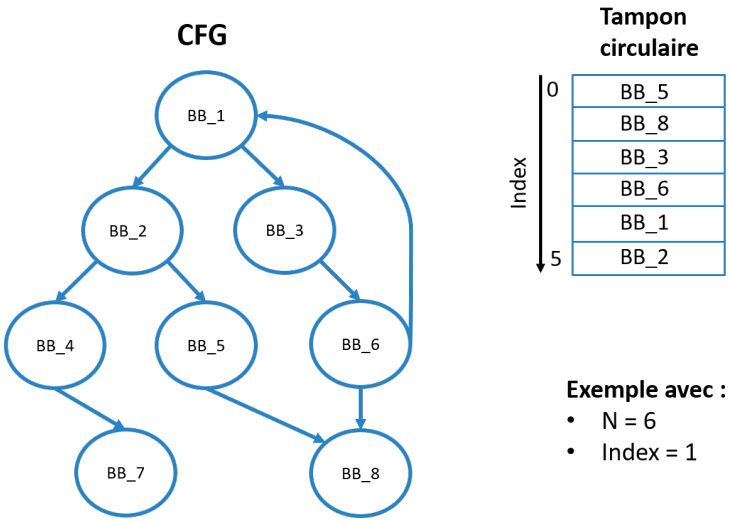

FIGURE 5.9 – *Cas d'exemple de vérification.*

La première étape de la vérification est de récupérer le tampon circulaire. Comme nous sommes sur un tampon circulaire, il est aussi nécessaire de récupérer un index pour savoir où

en est le remplissage, et ainsi savoir quel est le dernier BB emprunté. La figure 5.9 propose un exemple: un CFG et un tampon circulaire contenant l'enchaînement des derniers blocs de base, avec une profondeur N = 6, et un index de départ à 1. Nous proposons d'analyser le pseudocode du programme de vérification que nous avons implémenté (algorithme 5) avec ce cas d'exemple.

Algorithme 5: Pseudocode de la tâche de vérification.

```
   Input: index, tampon, CFG
 1 addr_current = tampon[index] ;
 2 BB_current = CFG(addr_current) ;
 3 if BB_current == None then
 4 |   alert();
 5 |   return-1;
 6 end
 7 if index == 0 then
 8 |   prev_index = N-1;
 9 else
10 |   prev_index = index - 1;
11 end
12 for i=0;i<N;i++ do
13 |   prev_addr = tampon[prev_index] ;
14 |   exist = False ;
15 |   for BB in BB_current.predecessors do
16 |   |   if BB.addr == prev_addr then
17 |   |   |   exist = True ;
18 |   |   |   BB_current = BB ;
19 |   |   |   index = index - 1 ;
20 |   |   |   if index < 0 then
21 |   |   |   |   index = N-1 ;
22 |   |   |   end
23 |   |   |   if index == 0 then
24 |   |   |   |   prev_index = N-1;
25 |   |   |   else
26 |   |   |   |   prev_index = index - 1;
27 |   |   |   end
28 |   |   |   break ;
29 |   |   end
30 |   end
31 |   if exist == False then
32 |   |   alert() ;
33 |   |   return-1 ;
34 |   end
35 end
36 return 1 ;
```

L'idée principale est de vérifier que le chemin parcouru existe dans le CFG. La recherche est simple. On récupère l'adresse du dernier bloc de base emprunté à partir du tampon circulaire (ligne 1). À partir de cette adresse, nous pouvons vérifier qu'elle est bien présente dans le CFG (ligne 2). Si on se réfère à notre exemple, l'index de départ étant 1, nous récupérons le BB_8. Si le bloc de base n'est pas trouvé dans le CFG, alors on est forcément sous attaque (ligne 3

à 6). Si un BB est trouvé, alors ça ne veut pas forcément dire que le programme est dans son fonctionnement normal. Lors d'une attaque par réutilisation de code, les BB du programme sont bien exécutés, mais dans un ordre différent de celui du CFG. Il faut donc continuer la recherche. Les lignes 7 à 11 initialisent les index de recherche pour récupérer la prochaine adresse dans le tampon circulaire. On rentre ensuite dans une boucle de complexité O(N) (ligne 12 à 35). Cette boucle vérifie simplement que l'adresse suivante du tampon correspond bien à l'un des prédécesseurs du bloc de base courant. Si ce n'est pas le cas, alors une alerte est levée. Par exemple, le BB_8 possède deux prédécesseurs: BB_5 et BB_6. Si l'adresse précédente dans le tampon circulaire ne correspond à aucun des deux prédécesseurs, alors une alerte est levée. Dans le cas contraire, la recherche continue tant que tout le tampon n'est pas parcouru (ou qu'une alerte est levée). En termes de complexité, l'algorithme est très léger avec deux boucles, une de complexité O(N) et l'autre de complexité O(M), avec

$$M = moyenne(nombre\_de\_prédécesseurs) \tag{5.1}$$

Nous obtenons donc une complexité finale de O(M*N), avec *M* généralement petit. Ce nombre est généralement égal à 2 voire 3 (dans le meilleur des cas), 2 représentant une condition *if*. *N* dépend de la granularité choisie, mais il semble inutile d'avoir des valeurs trop grandes. D'une part à cause du stockage nécessaire : si on est sur une architecture 32 bits, alors il faut réserver un tampon circulaire de *N**4 octets. D'autre part, si l'on valide au moins les 20 derniers blocs de base, il y a de fortes chances que le programme soit dans une exécution normale, ce qui peut se révéler faux si on vérifie seulement 2 ou 3 BB.

Au final, la tâche de vérification est facile à mettre en place. La complexité du programme permet de le déployer aussi bien sur une cible embarquée que sur un serveur. La question qui se pose est l'espace de stockage nécessaire pour faire la vérification. Outre le besoin de stocker les *N* derniers BB exécutés, si la vérification est faite directement sur le système embarqué, il est aussi nécessaire de stocker le CFG. Les coûts mémoires augmentent donc en conséquence et sont équivalents au stockage nécessaire pour les CFI logiciels ou matériels.

5.2.4 Conclusion

Ces travaux proposent une première étude et une vision sur la possibilité d'implémenter un outil visant à diagnostiquer le flot de contrôle d'un programme à la demande. Nous avons découpé l'outil de diagnostic en trois parties distinctes :

1. **La construction du CFG** est finalement très facile à mettre en place grâce aux différents outils existants. Nous avons comparé ces outils, et proposé angr comme meilleur choix. Nous avons également montré la facilité d'utilisation de angr. Bien que le CFG puisse comporter des défauts, notamment au niveau des sauts indirects, ils n'affectent pas la sécurité globale de la solution. Ces défauts génèrent simplement des faux positifs lors de la vérification, et l'alerte est finalement traitée comme une attaque.
2. **La partie 2 sur l'instrumentation** est plus complexe. En effet, nous n'avons pas pour le moment de solution idéale. Si une approche émulateur semble la plus facile à mettre en place, dans l'idée d'un prototype, c'est aussi celle qui impacte le plus les performances. Nous pensons qu'une attention particulière doit être portée sur cette partie, qui rejoint finalement les spécifications de CFI.
3. **La tâche de vérification** est simple à mettre en place, avec un algorithme d'une faible complexité. La solution peut aussi bien être déployée sur le système embarqué que sur un serveur, si ce premier possède assez de mémoire pour stocker le CFG.

Au final, outre la problématique restante pour récupérer les informations nécessaires à la vérification à travers l'instrumentation, la solution complète semble possible, à moindre coût. Néanmoins, il reste à évaluer rigoureusement le coût du diagnostic par rapport à la contremesure. Si la vérification est rapide car elle consiste simplement en un programme O(N), avec N petit, la solution demande un ajout de pénalités constant sur le système pour mettre en place l'instrumentation. C'est finalement ce coût en performance qui permettra de valider ou non l'approche.

Cependant, au-delà des problèmes évoqués, une autre question se pose quant à la fiabilité globale de la méthode. À travers cet outil, il est possible de s'assurer que le flot de contrôle du programme a bien suivi le CFG à un instant t. Le diagnostic est lancé après chaque alerte de l'étage de détection pour vérifier si c'est effectivement une attaque, ou un faux positif. Cependant, que se passerait-il si le diagnostic se trompe ?

- **Les faux positifs** du diagnostique (dire que c'est une attaque alors que non) consistent simplement en une erreur dans le CFG, et sont donc vraisemblablement possible. Dans ce cas, c'est la même constatation que pour l'étage de détection. Les pénalités subies par le système augmentent à cause du diagnostic qui se révèle inutile.
- **Les faux négatifs** (diagnostic négatif alors qu'une attaque est présente) sont plus rares et un seul cas est critique. C'est finalement le même cas mettant en difficulté les CFI: une attaque sur les données et non sur le flot de contrôle. C'est le principe expliqué par les auteurs de *Control Flow Bending* [122], qui visent les arguments des différentes fonctions pour changer le comportement du binaire sans en changer le flot de contrôle. Le diagnostic ne serait pas capable de détecter ce cas, alors que la méthode de détection le peut. Le diagnostic introduit alors une erreur indésirable dans la sécurité globale du système. Cela remet finalement en cause la méthode complète et une étude plus poussée doit être faite sur ce mécanisme de diagnostic, en étudiant possiblement les méthodes de *Data Information Flow Tracking* (DIFT) qui visent à combattre les attaques sur les données aussi bien que les attaques sur le flot de contrôle.

5.3 Étude de la boucle de tolérance aux attaques

5.3.1 Contexte

Maintenant que nous avons étudié l'étage de diagnostic, nous proposons d'évaluer la boucle complète de la tolérance aux attaques. Lorsqu'une attaque est détectée, il est possible qu'elle ait déjà commencé à corrompre le système. Cette hypothèse dépend fortement de la méthode de détection utilisée. Par exemple, les méthodes à bases de détection de signatures ou d'anomalies permettent à l'attaquant d'exécuter leur attaque pendant un instant qui dépend de la rapidité de détection. L'hypothèse de départ est donc de se demander : est-il possible que l'attaquant corrompe la chaîne du traitement de l'attaque ? En falsifiant le diagnostic ? En trompant la décision ? En annulant la contremesure ? En effet, si l'attaquant est capable d'exécuter un code malveillant pendant un court instant, il est possible qu'il cherche à exploiter la chaîne de confiance et donc la sécurité globale du système avant de lancer sa vraie attaque. Dans cette section, nous proposons d'évaluer la chaîne complète, en étudiant les comportements des méthodes de détection vues dans l'état de l'art combinées à l'utilisation de contremesures. Nous n'étudions pas la possibilité de contourner le mécanisme de détection, mais la possibilité de contourner le rétablissement une fois l'attaque détectée.

5.3.2 Hypothèses

Dans un premier temps, nous proposons d'analyser de plus près le comportement des CFI car ce sont ces méthodes qui détectent les attaques par corruption de mémoire avec le grain le plus fin. Nous avons vu qu'il existe plusieurs solutions pour mettre en place un CFI. Les solutions logicielles telles que Control Flow Guard [92] de Microsoft, kBouncer [93], ou le CFI proposé par llvm [95], s'exécutent sur le même processeur que le code qu'ils protègent. Ces méthodes vérifient les changements de flot de contrôle (branchement d'une condition *if* par exemple) avant de l'accepter. Si le branchement n'est pas autorisé, une alerte est levée avant qu'il soit exécuté. On les appelle des CFI synchrones. De ce fait, ces implémentations ne laissent pas de fenêtre vulnérable, car l'attaque n'est pas exécutée, même un court instant. D'autres implémentations sont matérielles, et dépendent d'un co-processeur (ou une alternative sur FPGA) [115, 113, 114, 118, 117]. Ces implémentations diffèrent dans leur manière d'implémenter la vérification du flot de contrôle, ou encore dans les éléments matériels utilisés. Cependant, dans tous les cas, c'est un élément externe au CPU qui doit l'informer qu'un code malicieux commence à s'exécuter. La contremesure (représentée par l'étage action sur la figure 5.1) se passe donc en deux temps, une communication entre le co-processeur et le CPU, puis une action pour stopper l'attaque sur le CPU.

Même si nous avons vu qu'il existe une multitude d'implémentations de CFI matériels, seulement quelques études parlent de la contremesure mise en place lorsqu'une attaque est détectée [114, 156, 117, 119]. Dans leurs travaux sur les graphes de surveillance, Mao *et al.* [114] proposent de lancer une interruption en cas de détection d'attaque depuis leur coprocesseur. Wahab *et al.* [119] proposent une implémentation de DIFT directement dans la partie reconfigurable et utilisent aussi une interruption pour notifier le processeur en cas d'alerte. Lee *et al.* [156] proposent une implémentation matérielle sur FPGA pour se défendre des attaques ROP, et utilisent aussi une interruption pour notifier le processeur en cas de détection positive. Quelle que soit l'implémentation, après détection de l'attaque par le CFI, le coprocesseur ou la partie reprogrammable lève une interruption pour notifier le processeur. Ce dernier est ensuite en charge d'arrêter le processus corrompu. Cependant, il peut y avoir une latence entre le moment où l'attaque change le flot de contrôle du binaire et le moment où le CPU est notifié. Ces latences peuvent être dues à la communication entre le CPU et le co-processeur pour éviter de mettre en pause le CPU à chaque vérification. Il en résulte que pour garder des pénalités de performances les plus faibles possibles, la vérification du flot de contrôle est parfois faite après l'exécution sur CPU. Ces CFI sont appelés des CFI asynchrones. Par exemple, Mao *et al.* [114] proposent de dériver un flux d'information composé d'un des éléments suivants: adresse, *opcode*, instruction de *load* et de *store*, le flot de contrôle ou encore un hachage de l'*opcode*. Le comportement attendu du programme est généré par analyse statique, et est ensuite stocké en mémoire. Au moment de l'exécution, le flux d'information est comparé au graphe de surveillance. En cas de non-concordance, on suppose qu'il y a eu une attaque et le processeur est interrompu (via une interruption).L'attaque est donc détectée en un à dix cycles d'instructions selon les éléments combinés. Rahmatian *et al.* [157] ont utilisé une machine à états finis pour détecter les séquences d'appel système non valides. L'observation est que le code malveillant doit invoquer des appels systèmes pour effectuer certaines des opérations nécessaires au lancement d'une attaque. Un modèle des séquences autorisées est dérivé du CFG et la vérification est implémentée sur un FPGA. La méthode a un impact nul sur les performances et un appel système non prévu peut être détecté en trois cycles d'instructions. Une interruption est ensuite appelée pour arrêter le programme.

Le parallèle peut être fait avec les méthodes indirectes qui détectent des signatures ou des anomalies. Comme ces méthodes détectent l'attaque de manière indirecte, à l'aide de comp-

teurs de performance par exemple, alors il est nécessaire que l'attaque commence à s'exécuter pour être détectée. La latence de détection dépend de la granularité de la méthode, mais il en résulte qu'un attaquant dispose d'une fenêtre (plus ou moins grande) pour essayer d'exploiter le système. Donc, à la différence des CFI où l'attaquant ne va disposer que de quelques instructions pour essayer de compromettre la chaîne de rétablissement, ici il peut disposer d'une fenêtre bien plus grande. Cependant, pour étudier la sécurité globale (du point de vue d'un attaquant), nous choisissons de nous placer dans les pires conditions (cas des CFI) pour l'attaquant et les meilleures pour le système.

5.3.3 Modèle de menace

```c
#include <stdio.h>
#include <string.h>

void fonction_vuln(char *arg)
{
    char tableau[64];
    strcpy(tableau, arg);
    printf("Copie de arg dans Tableau terminée.\n");
    return;
}

int main(int argc, char *argv[])
{
    if(argc != 2)
        printf("Un seul argument requis.");
    else
        fonction_vuln(argv[1]);
    return 0;
}
```

FIGURE 5.10 – *Programme vulnérable.*

Dans ce travail, nous supposons que le flot de contrôle du programme peut être attaqué et qu'une méthode de détection quelconque est mise en place pour protéger le programme. Pour ça, nous proposons un programme simple (Figure 5.10) qui contient une vulnérabilité: un dépassement de tampon sur la pile. Nous évaluons l'axe entier de la tolérance aux attaques, et nous posons deux contraintes :

1. D'une part, nous visons les méthodes de détection qui laissent à un utilisateur une fenêtre d'attaque. Elle permet d'exécuter au minimum une instruction, avant de détecter l'attaque et de lancer le cycle de rétablissement du système. On considère que les méthodes de détection sont idéales (100% de précision).

2. D'autre part, nous visons les méthodes de rétablissement et plus particulièrement de contremesure (représentée par l'étage d'action (figure 5.1) qui utilisent une interruption pour notifier l'attaque au CPU. Le CPU prend ensuite les dispositions nécessaires pour arrêter l'attaque, en tuant le processus par exemple.

La figure 5.11 introduit un processus basique de protection et d'attaque que nous allons utiliser pour tester et évaluer la robustesse de la sécurité. Tout d'abord, un CFG (ou un modèle du programme) est construit hors ligne à l'aide du binaire. À l'exécution, la tâche de détection compare le modèle avec le flot d'information dynamique récupéré du CPU. Quand le programme vulnérable débute (1), le CPU commence à exécuter les instructions du binaire.

Lorsque l'attaque commence, le flot de contrôle du programme change (2) grâce au dépassement de tampon sur la pile. L'attaque contient une charge malicieuse qui injecte une *ropchain* sur la pile, et redirige le flot d'exécution sur ce code malicieux (3). Après quelques instructions exécutées (1 ou plus), la tâche de vérification détecte l'attaque (4) et lance le processus de rétablissement (5). Connaissant maintenant les différentes parties du système, les hypothèses, et le modèle de menace, nous allons étudier si le cycle concernant la tolérance aux attaques contient des vulnérabilités, et dans le cas positif, si elles sont exploitables.

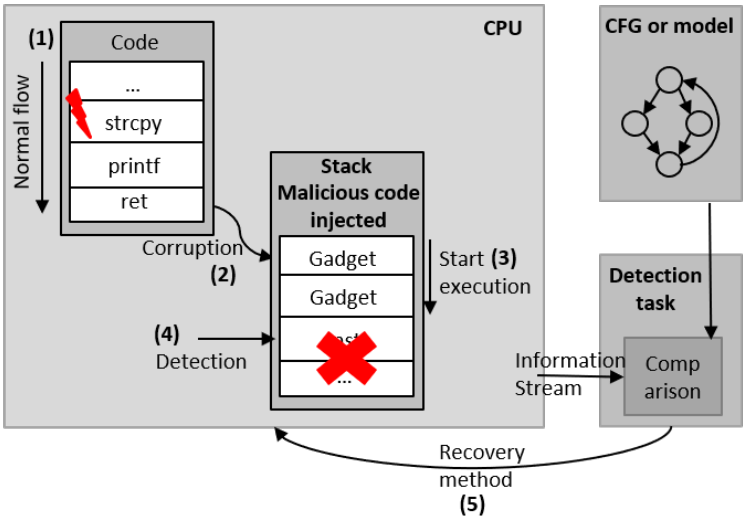

FIGURE 5.11 – *Modèle de menace.*

5.3.4 Analyse de la sécurité

Dans un premier temps, nous analysons les différentes parties pour voir si l'attaquant possède un moyen d'altérer le rétablissement du système. En outre, il s'agit d'étudier la notification envoyée, et voir s'il est possible de l'intercepter ou de la masquer. Il faut donc comprendre le fonctionnement des différentes interruptions qui existent, mais aussi les différents modes d'exécution de code dans lesquels peut être le CPU. Ces deux notions sont expliquées dans les parties 5.3.4.a et 5.3.4.b respectivement.

5.3.4.a Les interruptions

Il y a plusieurs types d'interruptions dans un système catégorisés en deux grandes familles :

- **Les interruptions logicielles :** qui sont divisées à nouveau en deux catégories :
  - **Les Exceptions :** Elles sont généralement causées par des programmes qui génèrent une erreur. Une écriture dans une zone mémoire non autorisée, une allocation mémoire insuffisante, ou encore un débordement dû à une opération arithmétique.
  - **Les Interruptions :** Elles sont généralement causées par le logiciel, par exemple un appel système depuis le noyau *(read, write, exec, etc)*.

Ces deux catégories, bien que déclenchées de manière différente, sont similaires dans leur fonctionnement. C'est-à-dire que lorsqu'une Exception ou une Interruption survient, c'est une fonction particulière qui est appelée, plus communément connue sous le nom de *routine de gestion d'interruption*.

- **Les interruptions matérielles :** Elles sont des signaux envoyés depuis des périphériques externes au processeur. Elles sont utilisées pour indiquer au CPU qu'un évènement extérieur s'est produit. De la même manière que pour les interruptions logicielles, une routine de gestion d'interruption est appelée pour lancer la tâche correspondante.

Parmi tous ces mécanismes, on retrouve une notion de priorité entre les interruptions; c'est-à-dire qu'une interruption peut être plus prioritaire qu'une autre et donc le noyau peut lancer/exécuter la routine de gestion d'interruption associée en priorité. Enfin, certaines interruptions sont dites *sécurisées*, c'est-à-dire qu'une attention particulière leur est portée vis-à-vis du système et n'importe quel code ne peut pas changer leurs réglages.

5.3.4.b Différents modes d'exécution de code

Avant d'utiliser les principes acquis pour essayer de corrompre le rétablissement du système, il est nécessaire de comprendre les différents modes d'exécution de code d'un CPU. Il existe plusieurs modes, mais nous allons nous intéresser seulement aux deux principaux :

- **Mode *Système ou Noyau (Kernel)* :** en mode *Système*, les droits d'exécution du code sont illimités. Le CPU a un accès total au matériel sous-jacent. Il peut exécuter n'importe quelle instruction et atteindre n'importe quelle adresse mémoire. Ce mode est généralement réservé à un OS ou à un code autonome (sans OS donc). Exploiter un code exécuté en mode *Système* laisse donc plus d'opportunités à un attaquant.

- **Mode *Utilisateurs* :** en mode *Utilisateur*, les droits d'exécution sont restreints. Par exemple, le programme ne peut pas directement accéder au matériel et doit passer par des API systèmes. C'est généralement le mode utilisé pour les applications d'un OS qui ne nécessitent pas un traitement particulier. En cas d'exploitation d'un programme en mode *Utilisateur*, un attaquant possède donc moins de marge de manoeuvre.

5.3.4.c Vers une vulnérabilité du cycle complet

Suite à ces analyses, aux hypothèses posées et au modèle de menace, nous proposons deux méthodes d'attaque logicielle sur l'interruption pour corrompre le système.

Chaque processeur peut ignorer une interruption (matérielle ou logicielle) en fixant un *flag*. Pour fixer ce *flag*, il existe des instructions spéciales: *st, cli* pour des architectures x86, x64, *cpsid* pour des architectures ARM. Cependant, ces instructions ne peuvent être exécutées que sous certaines conditions. Pour être capable de masquer toutes les interruptions du CPU, le code doit être exécuté en mode *Système*, autrement l'instruction sera considérée comme une instruction *nop* [158]. Cela veut dire que cette première méthode, que nous appellerons l'attaque *cpsid*, ne peut être effectuée que si le code appelant est privilégié, par exemple, une faille dans le noyau de l'OS, dans un pilote de matériel, ou un processeur exécutant du code autonome. Ces scénarios sont fréquents dans les systèmes embarqués car il n'est pas rare d'avoir des systèmes sans

OS ou utilisant divers pilotes pour gérer des capteurs, des radios ou encore des actionneurs. L'avantage de cette première méthode est qu'elle s'exécute en exactement une seule instruction.

Si le code s'exécute en mode *Utilisateur*, alors l'attaquant ne peut pas utiliser cette méthode. Néanmoins, utiliser ce *flag* n'est pas le seul moyen de masquer une interruption. Les processeurs possèdent des registres de contrôle qui permettent à l'utilisateur de configurer différents périphériques internes ou signaux. Une de ces configurations permet de gérer les interruptions. Donc, en utilisant ces registres de contrôle, un programmeur est capable de gérer les réglages des interruptions en changeant la priorité, ou alors, en masquant/démasquant ces dernières. Nous appellerons cette attaque "*w-w-w*" en référence à la méthode *write-what-where* qui permet d'écrire n'importe quelle valeur à n'importe quelle adresse. Cependant, pour changer ces réglages, le programmeur doit respecter quelques règles. Par exemple, si le code est exécuté en mode *Utilisateur*, alors il ne peut pas accéder à tous les réglages des interruptions. Si l'interruption est considérée comme sécurisée, seul un code exécuté en mode *Système* pourra changer les réglages. Cela veut dire qu'un attaquant pourrait ne pas être capable de changer les réglages de l'interruption si l'interruption est sécurisée. Dans le cas de l'architecture ARM, il faut aussi prendre en compte la Trustzone [159]. Au démarrage du système, la Trustzone peut être utilisée pour affecter des valeurs à des registres spécifiques, bloquant tout futur accès aux registres de contrôle. La Trustzone peut donc stopper cette deuxième méthode, mais elle n'existe malheureusement pas sur tous les processeurs et microcontrôleurs.

Tableau 5.2 – *Méthodes et conditions pour l'attaque.*

Mode CPU	**Méthode d'attaque**	**Trustzone (ARM)**	**Interruption sécurisée**	**Vulnérable**
Système	cpsid	X	X	Oui
Système	w-w-w	Oui	Oui	Non*
Système	w-w-w	Non	Oui	Oui
Utilisateur	cpsid	X	X	Non
Utilisateur	w-w-w	X	Oui	Non
Utilisateur	w-w-w	Non	Non	Oui

**: Vulnérable en cas d'attaque sur la Trustzone comme spécifié par [160].*
X: n'entre pas en compte.

Il existe donc deux méthodes, *w-w-w* et *cpsid*, pour venir masquer une interruption. Cependant, comme nous venons de voir, ces méthodes reposent sur des prérequis. Le Tableau 5.2 résume les différentes méthodes et conditions nécessaires pour que l'attaque soit un succès. En tenant compte du mode d'exécution du code sur le CPU *(i.e. Système ou Utilisateur)*, de la méthode d'attaque *(cpsid ou w-w-w)*, de la Trustzone (utilisée ou non) et de l'interruption (sécurisée ou non), nous pouvons déterminer si la configuration est vulnérable. Par exemple, si le mode d'exécution de code est *Utilisateur*, que la Trustzone n'est pas activée et que l'interruption n'est pas sécurisée, alors le système est vulnérable aux deux méthodes d'attaque: *cpsid* et *w-w-w*.

Bien qu'il y ait deux méthodes pour masquer l'interruption, l'attaque sera sensiblement identique. C'est-à-dire que pour les deux cas, une *ropchain* est utilisée. La différence va être dans l'utilisation des gadgets. Alors que la méthode *cpsid* n'a besoin de trouver qu'une seule instruction *cpsid I* dans le binaire, le deuxième cas demande un peu plus de travail mais les gadgets requis sont plus communs. La Figure 5.12 montre un exemple d'attaque utilisant la méthode *w-w-w* qui écrit une valeur arbitraire à une adresse arbitraire en utilisant une *ropchain*. Trois gadgets standards sont utilisés. On se souvient, comme expliqué dans la section 2.4, que les gadgets sont des fragments du code original, composés d'une à plusieurs instructions. Dans

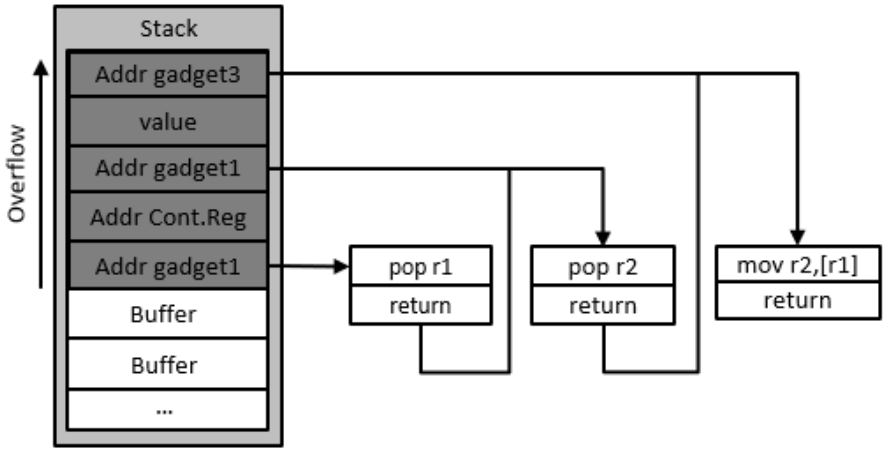

FIGURE 5.12 – *Concept d'une attaque ROP pour modifier les registres de contrôles.*

notre cas, ils contiennent deux instructions chacun. Cependant, l'attaque prend effet avant la dernière instruction de retour, donc l'attaque complète contient cinq instructions. La première instruction *"addr gadget"* écrase l'adresse de retour de la fonction de notre programme pour rediriger le flot de contrôle. L'attaquant a ensuite besoin de préparer deux valeurs, une correspondant à l'adresse du registre de contrôle et l'autre pour la valeur à régler. Ces valeurs sont à rechercher dans la *datasheet* du SoC et doivent être placé sur la pile du programme lors du dépassement de tampon. Dans ce cas, *"addr Cont.Reg"* est stocké dans le registre *r1* et *"value"* est stocké dans le registre *r2*. Le dernier gadget va permettre d'écrire cette valeur à l'adresse donnée.

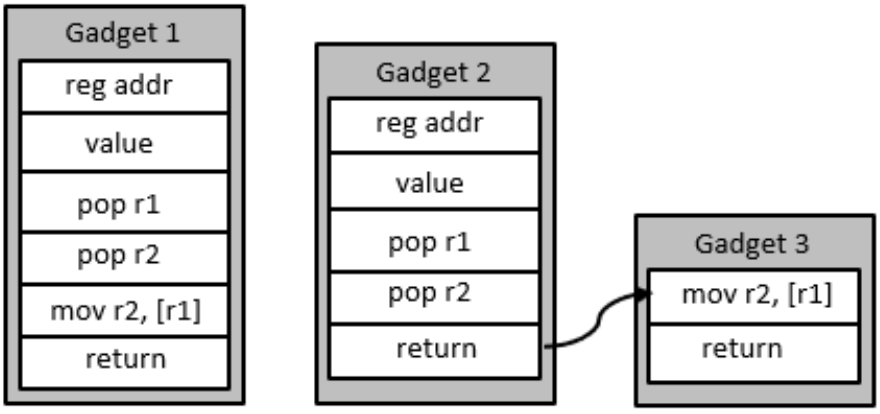

FIGURE 5.13 – *Gadget permettant de modifier les registres de contrôles.*

La Figure 5.13 donne différentes possibilités d'implémentation de l'attaque *w-w-w*. Par exemple, le Gadget 1 contient seulement trois instructions pour mettre en place l'attaque: deux instructions *pop* et une instruction *mov*. Alors que le chaînage du Gadget 2 et du Gadget 3 va contenir quatre instructions: deux *pop*, un *mov* et un *ret*. Donc il est possible de trouver différentes combinaisons de gadget pour réduire le nombre d'instructions nécessaires à l'attaque.

5.3.5 Implémentation de l'attaque

Maintenant que nous avons mis en lumière une vulnérabilité possible sur les méthodes de tolérance aux attaques et plus particulièrement sur la partie contremesure lors du rétablissement du système, l'étape suivante est de tester l'exploitation. Pour bien comprendre le fonctionnement de la vulnérabilité, un scénario d'attaque classique est analysé pour essayer de compromettre le système (système décrit dans la partie modèle de menace). L'idée est d'attaquer le moyen de communication de la contremesure pendant l'intervalle de temps où l'étage de surveillance n'a pas encore détecté l'attaque. Ainsi, le système ne peut pas revenir dans un état connu malgré la détection, car la contremesure n'est pas appliquée et la chaîne de rétablissement est rompue. Cette section décrit la mise en place d'un prototype sur Zybo. Les deux prochaines sous-sections vont respectivement présenter les différentes briques mises en place pour l'attaque puis le déroulement de l'attaque.

5.3.5.a Prototype

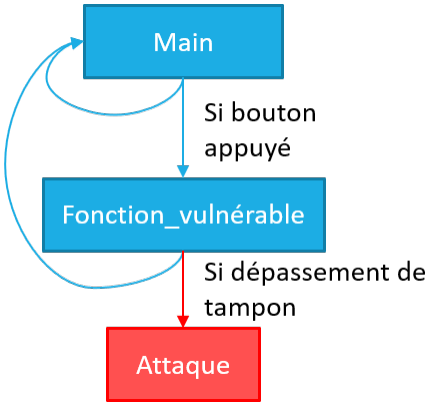

FIGURE 5.14 – *Redirection du flot de contrôle lors de la démonstration.*

Pour la démonstration, nous réutilisons la plateforme définie dans le chapitre 2, et plus particulièrement l'environnement développé sur la carte Zybo, sans la partie OS. Pour faire notre prototype, nous avons développé une application dans la partie Logique Reprogrammable (PL) visant à simuler un mécanisme de détection. Une application autonome est également exécutée par un CPU. Nous avons choisi une démonstration sans OS pour faciliter l'explication, le comportement étant strictement le même qu'avec un OS. De plus, le code est naturellement exécuté en mode *Système* pour permettre la mise en oeuvre des deux attaques présentées dans la section 5.3.4.c. Son rôle est expliqué à travers l'algorithme 6 et possède deux fonctionnalités principales. La première est de faire clignoter une LED tandis que la seconde est de déclencher une attaque par dépassement de tampon lors d'un évènement externe. Pour cela, une LED et un bouton-poussoir sont d'abord configurés lors de l'initialisation (ligne 7-8). Le programme principal rentre ensuite dans une boucle infinie (ligne 9). Cette boucle fait clignoter la LED (ligne 10-13). Ensuite, le programme lit la valeur actuelle du bouton pour appeler une fonction, vulnérable à une attaque par dépassement de tampon si la valeur est vraie (ligne 14-17). Cette fonction copie l'argument dans un tampon sur la pile (ligne 2-3). Si l'argument est plus grand que le tampon, alors un dépassement de tampon a lieu, et le flot est redirigé. Dans notre exemple, il est redirigé sur la fonction *Attaque*. Donc, au lieu de retourner dans la boucle infinie, le programme va venir afficher *"Flot redirigé"* (ligne 22-23) et arrêter le programme (donc éteindre la LED). Ce mécanisme est expliqué dans la figure 5.14.

Algorithme 6: Programme autonome pour la démonstration de l'attaque.

```
1  Function Fonction_vulnérable(arg):
2      tableau = int[50] ;
3      strcpy(tableau, arg);
4      return;
5  ;
6  Function Main():
7      gpio_led = init_gpio_led();
8      gpio_bouton = init_gpio_bouton();
9      while 1 do
10         led_on(gpio_led);
11         sleep(1);
12         led_off(gpio_bouton);
13         sleep(1);
14         valeur_bouton = lire_valeur_bouton();
15         if valeur_bouton then
16             Fonction_vulnérable(arg_prédéfini);
17         end
18     end
19     return 0 ;
20 ;
21 Function Attaque():
22     printf("Flot redirigé");
23     exit();
```

La figure 5.15 illustre l'implémentation du prototype sur Zybo. Un seul CPU est utilisé pour exécuter l'application. Les périphériques externes sont la LED et le bouton-poussoir utilisés par l'application. Le bouton-poussoir est connecté au Contrôleur d'Interruption Général (GIC). Le GIC est connecté au CPU ainsi qu'à la PL pour permettre la répartition des interruptions. Pour simplifier l'attaque et construire la *ropchain*, les gadgets nécessaires sont directement embarqués dans l'application. Pour finir, nous apportons quelques éclaircissements sur le simulateur de détection d'attaques. Pour démontrer l'attaque, nous n'avons pas besoin d'utiliser une méthode de détection spécifique comme un CFI par exemple. La vulnérabilité trouvée ainsi que l'attaque proposée ciblent n'importe quelles méthodes de détection à partir du moment où elles respectent nos hypothèses. Le comportement du simulateur est expliqué dans l'algorithme 7. Lorsqu'on appuie sur le bouton, un compteur démarre directement pour représenter la latence de la détection (ligne 3). La durée du compteur prend en compte le temps d'exécution de la fonction vulnérable ainsi que les trois instructions maximum après l'attaque (ligne 2). Enfin, l'interruption est envoyée lorsque le compteur est terminé (ligne 3). Ces propriétés ne changent en rien le comportement de l'attaque et du système de protection global. Elles permettent simplement de valider le concept de manière générale.

5.3.5.b Test des attaques

Dans cette section, nous testons les deux méthodes d'attaques sur le prototype. La figure 5.16 illustre le déroulement. Quand le bouton-poussoir est activé, un signal est envoyé au CPU pour débuter l'attaque (1). Au même moment, le simulateur de CFI démarre un compteur (1bis)

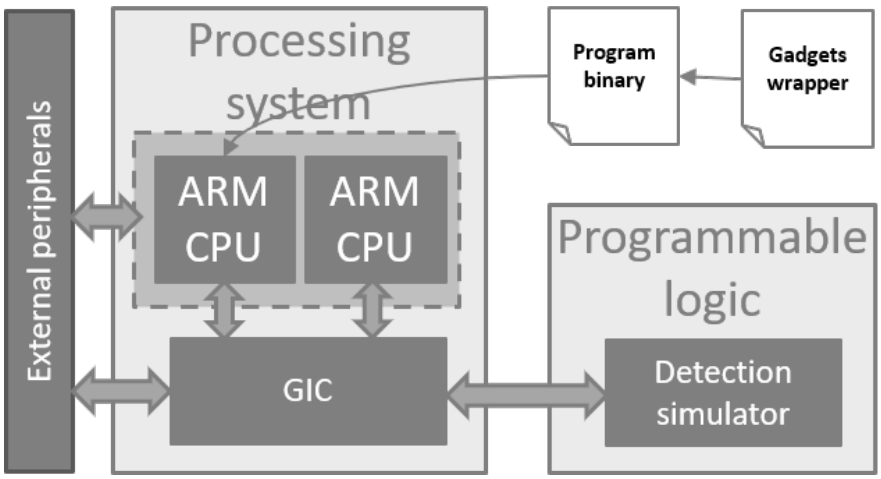

FIGURE 5.15 – *Implémentation du prototype de test.*

Algorithme 7: Application s'exécutant sur le FPGA pour simuler un mécanisme de détection.

```
1 while 1 do
2     if bouton then
3         Démarrage_compteur();
4         Attendre_compteur();
5         Lancer_interruption();
6     end
7 end
```

pour le temps de trois instructions et déclenche une interruption à la fin de ce compteur pour arrêter le CPU et retrouver un état connu.

Dans le cas de la méthode *cpsid*, la première instruction exécutée est *cpsid I* (2). Nous insérons volontairement une dizaine d'instructions *nop* à la suite pour perdre du temps entre l'attaque de l'interruption (2) et le véritable exploit (5). Cela permet d'être certain que nous n'avons pas de problème de synchronisation mais que l'attaque a effectivement fonctionné. Comme notre simulateur détecte un changement après trois instructions, le changement de flot de contrôle est détecté après deux instructions *nop* (3). Le simulateur lève ensuite une interruption (4), mais celle-ci est ignorée par le CPU à cause de l'étape (2). Comme l'interruption n'est pas reçue, l'exécution du programme n'est pas arrêtée, et l'attaque continue (5) et stoppe définitivement la LED. Les expérimentations ont montré que le simulateur essaie bien d'interrompre le CPU après trois instructions exécutées car en l'absence d'attaque, la LED ne s'arrête pas. Donc un attaquant peut bien contourner toute la politique de sécurité d'un système en s'attaquant directement à sa contremesure.

Nous avons aussi mis en place la seconde méthode basée sur les registres de contrôle. Le concept est exactement le même, excepté qu'il faut remplacer le gadget *cpsid* par un gadget *w-w-w* pour écrire une valeur spécifique dans un registre de contrôle donné pour masquer l'interruption. Le registre à régler dépend fortement du type d'interruption utilisé, mais la méthode reste la même. Dans notre cas, la méthode la plus sûre pour trouver l'adresse et la valeur à écrire est de trouver ces informations dans la datasheet du Zynq-7000 [62].

Cependant, dans ce scénario, la Trustzone de ARM peut empêcher l'attaque. Au démarrage

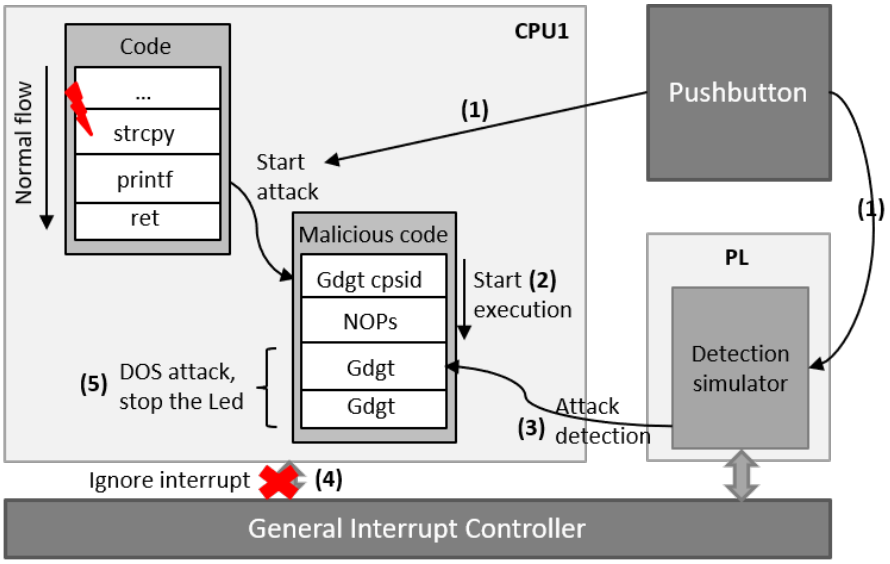

FIGURE 5.16 – *Implémentation de l'attaque.*

du système, un registre particulier peut être écrit pour bloquer tout futur droit d'écriture aux registres de contrôles après le démarrage du CPU. Donc une bonne utilisation de la Trustzone doit permettre à un utilisateur d'endiguer cette deuxième attaque, que le code soit exécuté en mode *Système* ou en mode *Utilisateur*. Néanmoins, comme démontré dans [160], la Trustzone peut être attaquée. Benhani *et al.* présentent des travaux pour compromettre la Trustzone dans des environnements SoC. Plus précisément ils proposent des modifications malicieuses des bus AXI ou de l'AXI *Interconnect* lors de la phase de conception pour permettre un contournement de la Trustzone à l'exécution. Ce scénario implique que l'attaquant ait dans un premier temps compromis le système de *build* des développeurs. De plus, comme nous l'avons dit, la Trustzone n'est pas disponible sur tous les processeurs et microcontrôleurs. Donc même si elle peut permettre dans certains cas de contrer la deuxième attaque, nous ne pouvons pas nous reposer entièrement sur cette protection.

5.3.6 Conclusion

Suite à ces travaux, nous avons pu identifier une vulnérabilité au niveau de l'enchaînement détection/contremesure. Nous avons montré qu'utiliser une interruption pour notifier le CPU une fois l'attaque détectée n'est pas sûr et que cela peut être contourné sous certaines conditions. Nous avons défini deux méthodes d'attaques pour permettre ce contournement, basées sur un masquage de l'interruption. Nous avons ensuite mis en place un prototype qui a permis de valider la vulnérabilité et de montrer des exemples d'exploitation. Même si nous avons défini deux méthodes d'attaque, les conditions pour les mettre en place sont différentes. La première, *cpsid*, a besoin que le code tourne en mode *Système* pour pouvoir fonctionner. Ce cas d'usage cible donc plutôt des problèmes dans le Noyau, les pilotes ou encore les codes autonomes. La seconde, *w-w-w*, a une portée plus large, mais peut être facilement stoppée par la Trustzone. Néanmoins, la Trustzone n'étant pas disponible dans tous les processeurs ou microcontrôleurs, l'attaque peut rester possible. Il en résulte que la chaîne de sécurité complète peut être compromise. C'est pourquoi il est très important de bien étudier le système dans son ensemble. Bien que l'étage de surveillance soit un instrument fiable qui détecte bien l'attaque, les moyens mis en oeuvre pour rétablir le système après la détection ne sont pas suffisants. Cette vulnérabilité est principalement due à l'association de deux méthodes, qui se veulent sécuri-

sées et efficaces lorsqu'elles sont évaluées indépendamment l'une de l'autre. Mais lorsqu'on combine la latence de la détection avec la contremesure qui doit notifier l'attaque au CPU (à travers une interruption), alors on obtient une vulnérabilité critique qui permet de corrompre le système.

Nous avons donc deux choix principaux pour garder un système sûr. Le premier est d'utiliser des méthodes de détection qui ne laissent pas d'opportunité à l'attaquant. Par exemple, le CET d'Intel [116] est un CFI matériel synchrone basé sur une modification du processeur et n'est donc pas vulnérable à notre modèle de menace. Dans la même idée, le Control Flow Graph de Windows [92], qui est une implémentation de CFI qui valide les branchements avant de les exécuter, n'est donc pas vulnérable. Cependant, si on prend les méthodes de détection d'anomalies ou de signatures, qui reposent sur les valeurs des compteurs de performances pendant l'exécution, alors il n'est pas possible de détecter l'attaque avant que celle-ci ne commence à corrompre le système. Il est donc nécessaire de s'orienter vers un deuxième choix: changer la contremesure utilisée pour qu'elle ne soit pas altérable, quelle que soit l'attaque. Pour ça, des travaux supplémentaires ont été effectués et sont présentés dans la partie suivante.

5.4 Méthode fiable de rétablissement

5.4.1 Contexte

Dans la section précédente, nous avons souligné une vulnérabilité sur les méthodes de tolérance aux attaques et plus particulièrement de la combinaison de l'étage de surveillance avec la chaîne de rétablissement. Nous avons montré que malgré la capacité à détecter une attaque, il est possible pour un attaquant de contourner la contremesure, et de ce fait, exécuter son attaque. Comme nous voulons garder la possibilité pour l'outil de surveillance de détecter une attaque après que cette dernière ait commencé à corrompre le système, il est nécessaire de travailler sur l'axe de la contremesure pour trouver une solution au problème. De ce fait, il faut que le dernier étage, qui représente l'action à appliquer pour rétablir le système, soit très fiable et non corruptible. De plus, si le module de détection est implémenté en dehors du processeur principal, pour des raisons de performances, alors il faut que la contremesure soit applicable par un élément externe. Une autre contrainte est de prendre en compte les fonctionnalités déjà présentes sur un processeur/microprocesseur, pour que la solution soit facilement déployable, mais aussi facilement adaptable pour différents systèmes. Nous ne voulons pas que la solution repose sur de nouvelles briques matérielles qui freineraient son déploiement. Enfin, nous voulons garder la possibilité d'exécuter l'outil de détection pour un coprocesseur ou un FPGA, pour limiter les impacts de performance, lorsque l'implémentation est possible. Nous avons donc pris en compte ces différentes contraintes pour concevoir ce nouveau mécanisme. Cette section est organisée comme suit. Dans un premier temps, une étude rapide de l'architecture basique d'un système sur puce (SoC) et des différents moyens de communication entre CPU est faite. Dans un second temps, une solution basée sur la réinitialisation et le redémarrage de sous-parties d'un système est proposée.

5.4.2 Aspects théoriques et hypothèses

L'idée principale de ce travail est de proposer une solution adaptable sur SoC qui permette de communiquer de manière sécurisée entre deux éléments internes. La figure 5.17 donne un

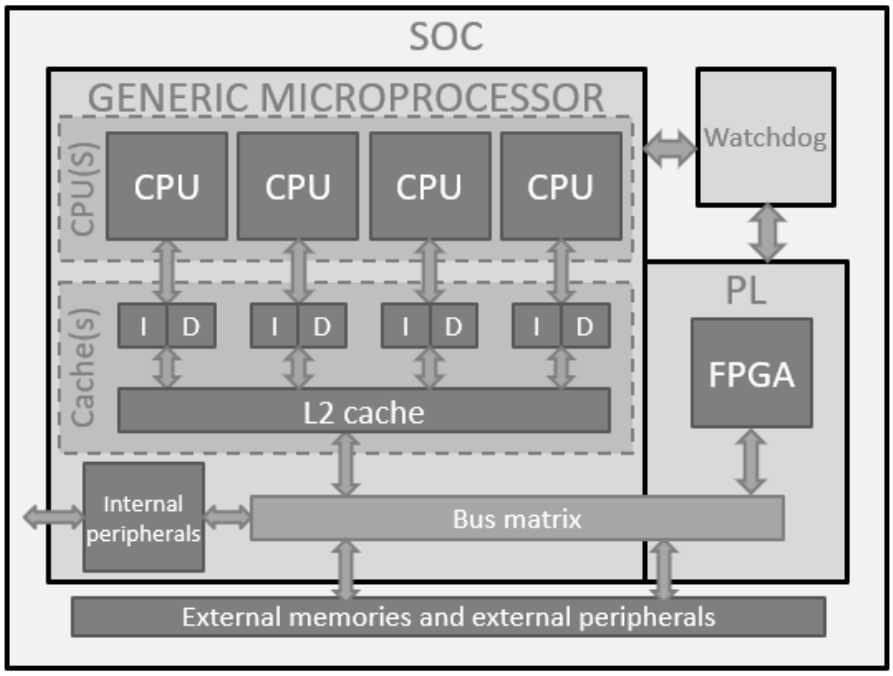

FIGURE 5.17 – *Architure SoC simplifiée.*

exemple de SoC. Il est composé de plusieurs CPU, de différents niveaux de cache, de périphériques internes et externes, mais aussi dans certains cas d'une partie matérielle reprogrammable (PL) contenant un FPGA. Tous ces éléments internes communiquent à travers des bus de données. Connaissant cette architecture de base, il faut décomposer chaque élément pour voir les possibilités de communication qu'il offre. Nous avons vu que les interruptions en sont un moyen, mais qu'elles n'offrent pas toujours une sécurité optimale. Un autre moyen serait d'utiliser les adresses mémoires partagées. Cependant les réglages de ces zones mémoires partagées sont similaires aux réglages des interruptions et seraient donc vulnérables de la même manière à la méthode d'attaque *w-w-w*. Deux autres méthodes intéressantes méritent de s'y attarder: les chiens de garde (Watchdog) et les méthodes de réinitialisation complètes ou partielles. Nous allons les étudier respectivement dans les deux prochaines sous-sections.

5.4.2.a Chien de garde

Un chien de garde est un élément matériel ou logiciel utilisé en électronique numérique pour s'assurer que le système ne reste pas bloqué dans un état. Le chien de garde a pour objectif de réinitialiser l'élément fautif. Lorsqu'il est logiciel, il s'agit en général d'un compteur qui est régulièrement remis à zéro. Ce compteur est représenté par un registre qui est mis à jour régulièrement via une interruption. Si le compteur dépasse une valeur donnée, alors un redémarrage du système est ordonné. Cependant l'implémentation logicielle du chien de garde sur le CPU qu'il protège est une mauvaise pratique. En effet, rien ne garantit que le chien de garde sera toujours en état de fonctionnement correct une fois l'attaque débutée. Lorsque les chiens de garde sont matériels, c'est généralement un circuit numérique qui va compter. Ce compteur est donc immuable face à une attaque, et si l'application ne le met pas à jour, alors, une interruption est générée pour lancer l'action prévue à cet effet. Cependant, l'interruption est sensible à la même attaque présentée dans la section précédente. Néanmoins, le chien de garde est aussi capable de directement demander une réinitialisation du système. Cette réinitialisation, une fois lancée, ne peut pas être altérée, car une liaison matérielle est faite entre le chien de

garde et le *reset* du processeur. Cette dernière solution est donc très intéressante dans le cadre de notre cahier des charges. Néanmoins, l'utilisation de cette technique dans un système sur puce peut poser des problèmes. Typiquement, il existe plusieurs chiens de garde dans un SoC :

- **Le chien de garde au niveau système :** c'est un élément à part entière du SoC et il est seulement capable de redémarrer le système entier. Cette méthode ne peut pas être altérée. Cependant, dans un SoC, on retrouve différentes parties. Il y a plusieurs processeurs et/ou une partie matérielle reprogrammable, et réinitialiser, puis redémarrer le système global, n'est pas une approche optimale puisque cela va amener des pénalités en performance lourdes.

- **Le chien de garde privé :** c'est un élément privé au CPU, qui ne peut redémarrer que ce dernier. Il permet donc de redémarrer seulement la partie défaillante. Malheureusement, les chiens de garde privés ne sont pas accessibles depuis un élément externe.

Ces éléments ne sont finalement pas utilisables dans notre solution. Il faut donc trouver un moyen de déclencher un redémarrage d'une partie spécifique depuis un élément externe. C'est ici que la réinitialisation logicielle devient intéressante.

5.4.2.b Réinitialisation

De la même manière que pour la gestion des interruptions, il existe des registres de contrôle qui permettent de réinitialiser des sous-parties d'un SoC. Plus connue sous le nom de réinitialisation logicielle, cette méthode permet à un élément externe privilégié de lancer un redémarrage du système complet ou d'une partie spécifique. Il est à noter qu'un élément ne peut pas s'auto-redémarrer. Cette propriété est très intéressante, car cela restreint les possibilités de l'attaquant pour corrompre le redémarrage. En effet, la partie attaquée n'est pas (et ne peut pas être) en charge de traiter l'ordre de réinitialisation, c'est l'élément externe qui l'applique à travers des écritures de registres. De plus, il n'existe aucun moyen de bloquer l'écriture à ces registres lorsque l'utilisateur en possède les droits. Il en résulte que redémarrer est plus sécurisé que notifier le système par une interruption. Cependant, le temps mis par le système pour retrouver un état connu et sûr s'en retrouve fortement augmenté. Aussi, comme nous allons le voir, redémarrer seulement une partie précise dans un système avec plusieurs CPU demande des efforts importants de développement. De plus, la technique n'est pas directement adaptable d'un système à l'autre, et même si elle reste similaire d'un SoC à l'autre, il est généralement nécessaire de faire quelques ajustements.

5.4.3 Implémentation

Avant de passer à l'évaluation de la solution, nous apportons quelques modifications à notre prototype pour passer sur un système multiprocesseur. Dans un deuxième temps, nous expliquons comment se passe une réinitialisation logicielle sur un système SoC en prenant le Zynq-7000 comme exemple (même système que précédemment). Une fois la réinitialisation faite, que ce soit de CPU à CPU ou de PL à CPU, il est nécessaire de redémarrer ce dernier. Nous étudions cette possibilité dans les deux cas. Nous proposons également d'évaluer la sécurité de la méthode, pour s'assurer de sa fiabilité et de sa robustesse.

5.4.3.a Système à plusieurs processeurs

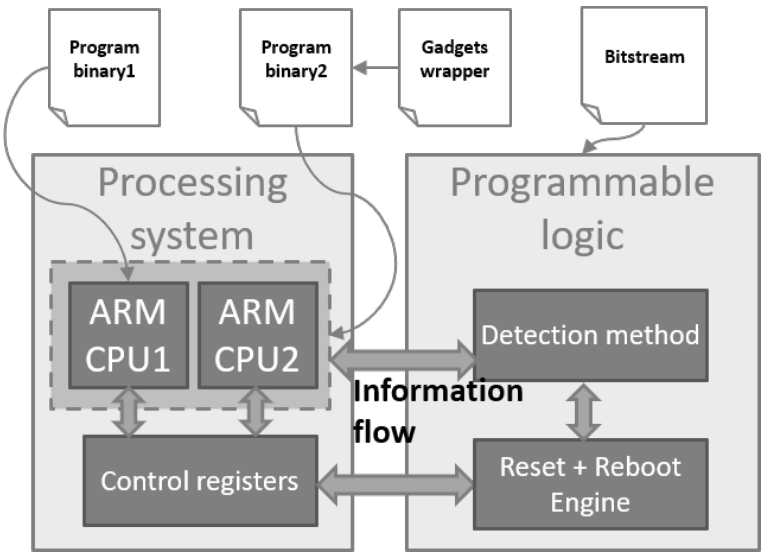

FIGURE 5.18 – *Prototype pour l'implémentation de la contremesure.*

Le second prototype proposé (Figure 5.18) comporte des similitudes avec celui présenté dans la partie précédente (Figure 5.10). Cette fois, deux CPU sont utilisés dans la partie *Processing System* pour prouver qu'il est possible de n'en redémarrer qu'un. Le mécanisme de détection est toujours présent dans la partie reconfigurable (PL) et la contremesure précédemment basée sur une interruption est remplacée par la méthode de réinitialisation + redémarrage. Nous proposons d'évaluer directement la méthode de PL à CPU, car plus complexe à mettre en place, et passer ensuite de CPU à CPU restera possible. Pour ce scénario, chaque partie exécute sa propre application :

- **Le CPU1** (notre CPU principal) est en charge de démarrer le CPU2, puis de faire clignoter une LED. Dans un système multiprocesseur, le CPU principal est en charge de démarrer tous les autres CPU. Son programme est présenté dans l'algorithme 8.
- **Le CPU2** exécute le même programme présenté dans la partie précédente (Algorithme 6), qui représente un programme vulnérable faisant clignoter une LED.
- **La PL** exécute la méthode de protection dont le comportement est décrit par la fonction *Fonction_detection* de l'algorithme 9. Son rôle est de protéger le code s'exécutant sur le CPU2. Nous proposons également d'ajouter une LED clignotante gérée par la PL (Algorithme 9, *Fonction_LED*) pour des besoins de démonstration.

Les LED sont présentes dans la démonstration pour illustrer quelles sont les parties qui sont en train de s'exécuter correctement. Lorsqu'un élément est réinitialisé, la LED s'arrête de fonctionner un instant, alors que si l'attaque réussit, la LED s'arrête de fonctionner définitivement. Ainsi, il est facile de visualiser si notre solution est capable d'arrêter une attaque et si elle est capable de cibler seulement la partie fautive.

Algorithme 8: Programme du CPU1.

```
1 démarrer_cpu2();
2 while 1 do
3 |   led_on(gpio_led);
4 |   sleep(1);
5 |   led_off(gpio_bouton);
6 |   sleep(1);
7 end
```

Algorithme 9: Programme VHDL dans la PL pour la démonstration.

```
1  Function Fonction_detection(bouton):
2  |   while 1 do
3  |   |   if bouton then
4  |   |   |   Démarrage_compteur();
5  |   |   |   Attendre_compteur();
6  |   |   |   Redémarrer_CPU2();
7  |   |   end
8  |   end
9  ;
10 Function Fonction_LED():
11 |   while 1 do
12 |   |   led_on(gpio_led);
13 |   |   sleep(1);
14 |   |   led_off(gpio_bouton);
15 |   |   sleep(1);
16 |   end
```

5.4.3.b Réinitialisation logicielle

Unlock SLCR	⟺	Write 0xDF0D in 0xF8000008
Assert CPU2 reset	⟺	Write 0x2 in 0xF8000244
Stop CPU2 clock	⟺	Write 0x22 in 0xF8000244
Release CPU2 reset	⟺	Write 0x20 in 0xF8000244
Start CPU2 clock	⟺	Write 0x0 in 0xF8000244
Lock SLCR	⟺	Write 0x767B in 0xF8000004

FIGURE 5.19 – *Séquence de réinitialisation logicielle.*

Nous avons vu les registres de contrôle du système dans la section 5.3.4.c avec les différents réglages des interruptions. Ils permettent à l'utilisateur d'interagir avec des sous-parties du SoC. Dans le cadre de ce travail, nous nous sommes intéressés aux registres qui permettent de lancer une réinitialisation. La méthode consiste à écrire une séquence de valeurs à des adresses précises. Cette séquence est définie dans la figure 5.19. Sur la partie gauche, on peut observer la

suite d'actions à effectuer, et sur la partie droite la valeur à écrire dans le registre correspondant pour réinitialiser le CPU2. Cette séquence, bien qu'étudiée pour le Zynq-7000, reste facilement adaptable sur d'autres SoC, le plus gros changement étant les valeurs et adresses définies. Ici, la première étape est de déverrouiller les registres de contrôles (SLCR) pour pouvoir écrire dedans. En effet, les registres de réinitialisation font partie des registres SLCR qui sont sécurisés par défaut et seul un élément privilégié peut les déverrouiller pour y accéder. Ensuite, il faut envoyer le signal de la réinitialisation et arrêter l'horloge du CPU ciblé. Une fois l'horloge à l'arrêt, il faut repasser le signal de la réinitialisation à zéro, et redémarrer l'horloge pour faire repartir le CPU. La dernière étape est de bloquer à nouveau l'accès aux registres SLCR pour éviter un accès non voulu ou malintentionné. Nous proposons de mettre en pratique ces concepts à travers une démonstration.

5.4.3.c Implémentation depuis la partie matérielle reprogrammable

Pour arriver à écrire la séquence de réinitialisation logicielle dans les registres SLCR depuis la PL, la première étape est de pouvoir accéder à ces registres depuis la PL. Pour cela, comme indiqué sur la figure 5.17, un bus de données peut être utilisé. Dans le cas du Zynq-7000, le bus qui connecte tous les éléments entre eux est un bus AXI4 [161]. La communication sur un bus AXI est basée sur un modèle maître/esclave. Donc pour pouvoir communiquer sur ce bus depuis le FPGA, il faut d'abord, implémenter une interface maître AXI4-LITE avec des droits d'accès suffisants pour accéder aux registres SLCR. Une fois cette interface créée, il faut pouvoir écrire la séquence de réinitialisation. Pour cela, nous avons choisi d'implémenter une machine d'états qui écrit cette séquence et permet de réinitialiser chaque CPU indépendamment depuis la PL. Cependant, une fois le CPU réinitialisé, il reste une étape importante qui est le redémarrage. Avant de savoir comment redémarrer le CPU, il faut comprendre comment se passe le démarrage. Lorsque le système est mis sous tension, le code dans la mémoire *bootROM* est exécuté. La *bootROM* contient le code de démarrage du fabriquant du SoC. Son rôle est de détecter et lancer le mode de démarrage adéquat, *i.e.* démarrage par carte SD, ou JTAG par exemple.

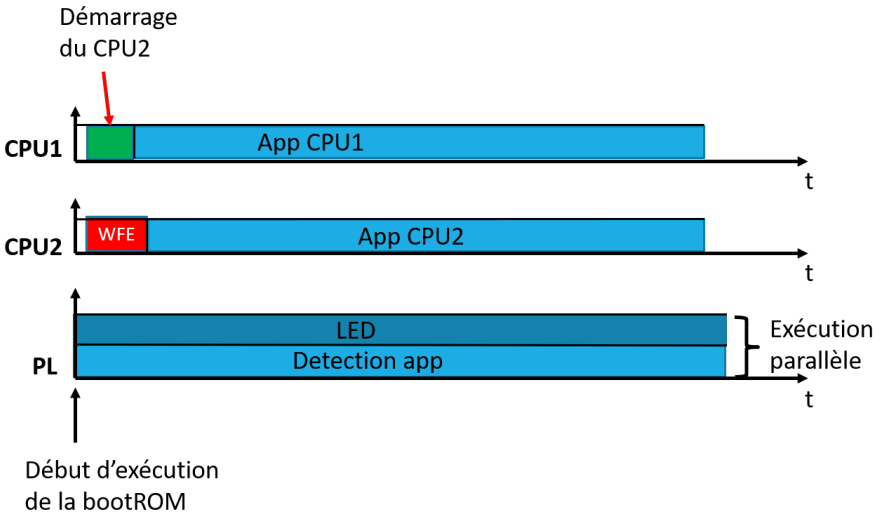

FIGURE 5.20 – *Étapes de démarrage des différentes applications.*

La figure 5.20 illustre les différentes étapes de démarrage du système. Dans un premier temps, le FPGA est programmé avec le *bitstream* (qui contient la description VHDL de l'algorithme 9). Ensuite le processeur principal (CPU1 dans notre cas) démarre, et le CPU2 entre en mode *wait-for-event* (WFE). Ce mode veut dire que le CPU2 est en attente. En d'autres termes,

il dort et attend qu'un évènement le réveille pour effectuer sa tâche. Au réveil, le CPU2 commence l'exécution du code à partir du pointeur contenu à l'adresse mémoire *0xFFFFFFF0*. Donc avant que le CPU1 envoie l'évènement pour démarrer le CPU2, il est nécessaire de préparer le programme correspondant. Pour ça, il faut simplement écrire le point d'entrée du programme du CPU2 à l'adresse *0xFFFFFFF0*.

Il faut savoir que lorsqu'un CPU est réinitialisé, son compteur de programme (PC) retourne à l'adresse *0x00000000* et commence à exécuter du code à partir de cette adresse. Cependant, cette dernière est partagée entre tous les CPU, il est donc impossible de prévoir sa valeur lors d'une réinitialisation. Le comportement du CPU sera donc non prédictible. Pour y remédier, il est nécessaire de préparer l'environnement avant de réinitialiser le processeur.

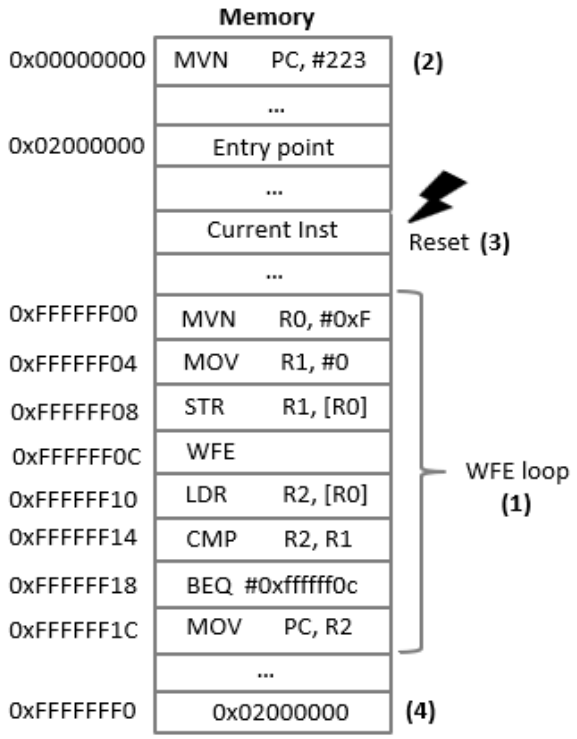

FIGURE 5.21 – *Séquence de redémarrage.*

L'idée est donc de repasser le CPU réinitialisé dans un état WFE. La figure 5.21 illustre le déroulement du redémarrage dans la mémoire. De la même manière que pour écrire la séquence de réinitialisation logicielle depuis la PL, la séquence de redémarrage du processeur est écrite à l'aide d'une interface maître AXI4-LITE et d'une machine d'état. Dans un premier temps, on écrit la boucle WFE (1) dans un espace mémoire réservé (de *0xFFFFFF00* à *0xFFFFFF1C*). La boucle a plusieurs rôles:

1. Initialiser l'adresse *0xFFFFFFF0* à la valeur 0. Cette action est réalisée par les instructions de *0xFFFFFF00* à *0xFFFFFF08*. Comme *0xFFFFFFF0* contient le point d'entrée du programme, l'idée est de l'initialiser une première fois pour être sûr que le CPU ne va pas exécuter une adresse inconnue.
2. Faire entrer le CPU dans le mode WFE (instruction en *0xFFFFFF0C*). Cette instruction est bloquante jusqu'à réception d'un évènement.
3. Lors d'un évènement, vérifier que la valeur de *0xFFFFFFF0* a changé. Si c'est toujours 0, alors le point d'entrée n'a pas été écrit correctement et le CPU retourne en mode WFE.

Sinon, le PC exécute le code à partir du point d'entrée contenu dans *0xFFFFFFF0* (instruction *0xFFFFFF10* à *0xFFFFFF1C*.

La deuxième étape est de faire entrer le CPU dans cette boucle après la réinitialisation. Pour cela, il suffit d'écrire l'adresse de la boucle à l'adresse *0x00000000* (2) pour directement rediriger le CPU sur la boucle une fois la réinitialisation lancée (3). La dernière étape est d'écrire le point d'entrée du programme à l'adresse *0xFFFFFFF0* (4) avant d'envoyer l'évènement pour sortir de la boucle et démarrer le CPU. C'est exactement le même comportement que lors du premier démarrage.

La figure 5.22 illustre le déroulement du redémarrage lors d'une attaque. Les applications sont dans un comportement normal, jusqu'à qu'une attaque soit détectée sur le CPU2 par l'application de détection. L'application exécutée par la PL rentre alors dans son étape de contremesure, et commence à écrire la boucle WFE et son point d'entrée dans la mémoire. La réinitialisation est ensuite lancée et passe le CPU2 en mode attente, pendant que la PL continue d'écrire le point d'entrée du programme du CPU2. Un évènement est ensuite déclenché, et le CPU2 redémarre.

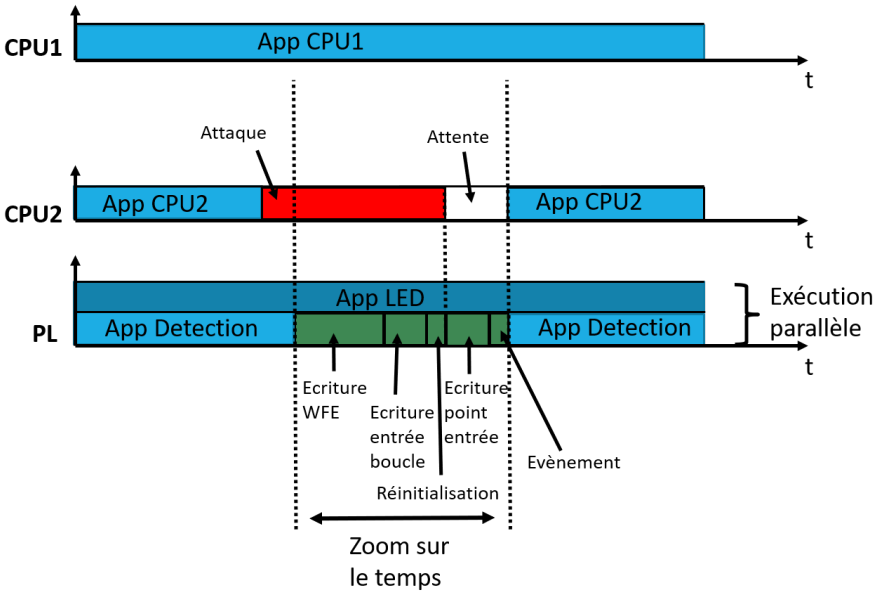

FIGURE 5.22 – *Étapes de redémarrage des différentes applications.*

Toutes ces séquences sont faites de manière atomique, c'est-à-dire que rien ne peut perturber le cycle ou la mémoire pendant son déroulement. Cette méthode nous garantit donc que le CPU est capable de redémarrer son programme après une réinitialisation.

5.4.3.d Évaluation de la méthode

Maintenant que le fonctionnement de la méthode de redémarrage depuis la PL est validé, la dernière étape est de tester et évaluer cette contremesure. Pour cela, nous avons repris le même scénario que dans la présentation de la vulnérabilité des interruptions en section 5.3.5.b. Dans ce cas d'usage, illustré par la figure 5.23, l'interruption est remplacée par le nouveau bloc de réinitialisation et redémarrage. La démonstration se déroule comme précédemment. L'attaque démarre lorsque le bouton est pressé (1). Elle vient rediriger le flot de contrôle du programme

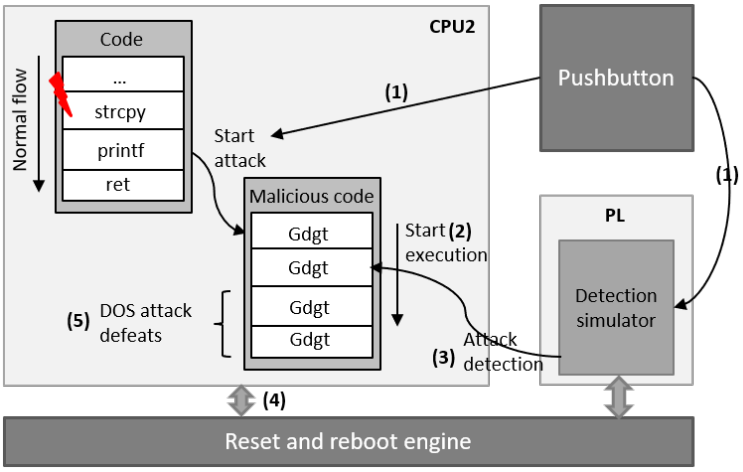

FIGURE 5.23 – *Protype de test de la nouvelle contremesure.*

(2) et le code malicieux commence à être exécuté. Au bout de trois instructions, l'attaque est détectée (3) et la contremesure est lancée(4). Cependant, dans ce cas, ce n'est pas une interruption qui est lancée, mais la nouvelle solution. Le processeur est réinitialisé et redémarré, et donc, l'attaque visant l'interruption échoue évidemment (5). Une question se pose toutefois: est-ce que cette technique n'introduit pas une nouvelle vulnérabilité ?

Il s'avère que l'accès aux registres SLCR demande une autorisation privilégiée. Pour attaquer la contremesure, le code exécuté doit avoir les droits du mode *Système*. Nous enlevons donc définitivement la menace venant d'un code exécuté en mode *Utilisateur*. Rien ne peut venir empêcher la PL (ou une autre partie) d'écrire dans les registres SLCR si elle en possède les droits. Il n'est donc pas possible non plus d'en empêcher l'accès. La seule solution serait d'essayer d'éteindre ou de redémarrer l'élément qui exécute la détection ou le rétablissement avant que la contremesure ne redémarre le processeur. Deux cas se profilent :

- Le cas où la granularité de la détection est **fine**. C'est-à-dire que l'attaquant ne dispose que de très peu d'instruction avant une détection et le déploiement de la contremesure. Il peut exécuter au maximum un gadget de type *w-w-w*, qui n'est pas suffisant pour appliquer une séquence de redémarrage. Au mieux, il pourrait seulement débloquer l'accès aux registres SLCR.
- Le cas où la granularité de la détection est **épaisse**. L'attaquant est capable d'exécuter plusieurs gadgets avant la détection et pourrait ainsi corrompre l'outil de détection. Ici, la solution est de bloquer l'accès à certaines plages d'adresses au CPU potentiellement malicieux. Par exemple, le CPU n'aurait pas l'autorisation d'accéder aux registres de réglages pour arrêter ou réinitialiser la PL, stoppant définitivement toute tentative d'attaques. Cette action peut facilement être faite à travers le gestionnaire d'adresses lors de la programmation du SoC, et n'impacte nullement les performances. Il en résulte qu'un attaquant ne peut plus mettre à profit la fenêtre d'attaque pour arrêter la contremesure.

Donc, le système peut retrouver un état sûr et connu quoi qu'il arrive lorsqu'une attaque est détectée. Cependant, il faut être conscient que cette méthode peut avoir de mauvaises répercussions en cas de faux positif de la part de l'étage de détection. Effectivement, réinitialiser et redémarrer un CPU complet peut amener des pénalités en performances conséquentes. Le coût en performance est cependant compliqué à estimer car cela dépend fortement du type d'application qui tourne sur le CPU. Cela peut aller d'un facteur nul pour une application autonome, à un facteur par 10 sur une application de type OS.

5.4.4 Conclusion

Suite aux travaux sur la vulnérabilité détaillée au cours de ce chapitre, nous avons travaillé sur l'élaboration d'une solution non altérable. Le cahier des charges était de déployer une solution qui puisse être utilisée dans un environnement SoC multi processeurs et facilement adaptable sur un système différent. Chaque élément devait être capable de cibler une partie spécifique du SoC pour être certain de retrouver un état sûr et connu sur ce dernier. Pour cela, nous avons étudié les différents moyens de communication internes au SoC. Nous avons vu que même s'il en existe une multitude (mémoire partagée, chien de garde, interruption), chacune de ces méthodes possède des limitations. Le périmètre s'en retrouve restreint, et aucune de ces caractéristiques ne pouvait subvenir aux besoins. Nous avons donc étudié une méthode de réinitialisation logicielle, couplée avec un redémarrage de la partie concernée. Une implémentation de cette solution a été pensée puis proposée afin de garantir que le système retrouvera un état sûr et connu après une détection d'attaque. Nous avons ensuite vu que malgré l'efficacité de la solution, elle peut être très lourde à mettre en place, *i.e.*, même si le déploiement de la solution est similaire d'un SoC à l'autre, son implémentation diffère et demande quelques efforts d'ingénieries supplémentaires. De plus, son exécution et le temps mis par le CPU pour retrouver son état sont bien plus longs que la solution basée sur une interruption. Le surcoût est compliqué à évaluer, car il dépend de plusieurs facteurs. Par exemple, si le processeur exécute un code autonome, alors le surcoût reste marginal, car traiter une interruption pour relancer le code ou réinitialiser le processeur revient sensiblement au même. Dans le cas d'un OS, type Linux, la pénalité devient plus conséquente, au lieu d'arrêter simplement un processus, il est nécessaire de relancer le système d'exploitation. Ce redémarrage peut être plus ou moins long, dépendant des services à démarrer ou encore de la distribution de l'OS. On gagne donc en sécurité, mais nous perdons fortement en performance, et encore plus si on prend en compte les faux positifs du système de détection qui vont demander un redémarrage complet pour rien. C'est pourquoi la solution peut et doit être améliorée. Une possibilité serait de pouvoir restaurer le système à une sauvegarde antérieure à l'attaque plutôt que de redémarrer toute la partie concernée.

5.5 Synthèse

Dans ce chapitre, nous nous sommes concentrés sur les méthodes de rétablissement. Dans un premier temps, nous avons étudié la possibilité d'implémenter une méthode de diagnostic pour éliminer les faux positifs de l'étage de détection. Nous avons proposé une méthode, inspirée des CFI, pour vérifier l'exécution du flot de contrôle d'un programme à la demande. Nous avons découpé le problème en trois parties, sans trouver de solution optimale pour l'instrumentation du programme permettant de sauvegarder le flot exécuté du programme. Des pistes ont été présentées, mais une étude plus poussée reste requise. De plus, nous avons conclu sur l'incapacité de la méthode à détecter des attaques sur les données du programme, introduisant ainsi une vulnérabilité critique dans le système. En effet, l'étage de surveillance étant capable de détecter ce type d'attaque, si le diagnostic se trompe, il fausse alors toute la chaîne de la tolérance aux attaques. Il reste alors à étudier des méthodes de diagnostic résistantes à ce genre d'attaques, tout en gardant à l'esprit que les pénalités du diagnostic doivent être inférieures à celles de la contremesure.

Dans un deuxième temps, nous avons évalué la combinaison de méthodes de détection et de rétablissement. Nous avons ciblé les méthodes de détection asynchrones qui laissent une fenêtre à l'attaquant pour exécuter quelques instructions malicieuses et ainsi corrompre le sys-

tème avant le déploiement de la contremesure. Nous avons apporté une première contribution en identifiant une vulnérabilité dans la chaîne. Les contremesures, qui utilisent une interruption pour notifier l'attaque au CPU, peuvent être contournées même si cette dernière est détectée correctement. Nous avons défini deux attaques de type ROP pour compromettre l'interruption. Bien que ces attaques aient besoin de pré-requis, *i.e.*, des gadgets, une fenêtre de temps entre la détection et l'attaque, des modes d'exécutions de code, ou même des réglages d'interruptions particuliers, cette étude prouve que l'utilisation d'une interruption pour notifier le processeur qu'il est corrompu peut être contournée. Donc, un état sûr et connu peut ne pas être retrouvé.

L'étude propose ensuite une seconde contribution pour venir améliorer la fiabilité et la sécurité de l'étage de contremesure. Il fallait comprendre les moyens à disposition d'un SoC, et identifier ce qu'il était possible de faire pour améliorer le rétablissement du système. Pour cela, nous avons étudié toutes les caractéristiques permettant de communiquer dans un SoC multiprocesseur et nous avons vu qu'aucune ne convenait à nos besoins. Soit la fonctionnalité était sûre mais non accessible depuis un élément externe, soit elle était altérable. Un compromis était d'utiliser la réinitialisation logicielle. Elle se devait d'être partielle pour ne pas amener trop de pénalités en performance et cibler seulement le CPU attaqué. Nous avons donc proposé une contremesure pour rétablir le système en s'assurant que la méthode ne puisse pas être altérée par l'attaquant.

Nous avons ensuite testé la nouvelle solution avec l'attaque et le modèle de menace de la première partie pour valider le concept. L'expérience a prouvé que notre solution est fonctionnelle pour contrer le premier modèle de menace. Puis, nous avons analysé, d'un point de vue de l'attaquant, comment corrompre cette solution. Nous avons étudié la capacité de l'attaquant à corrompre ou arrêter la réinitialisation quel que soit le mode d'exécution du code, et, quel que soit le nombre d'instructions exécutées par l'attaquant. Aucune menace n'a pu être identifiée. En contrepartie, la méthode de recouvrement demande plus de ressources de calcul et de temps pour revenir dans un état sûr et connu qu'une solution à base d'interruption. Même s'il est possible d'améliorer la solution en restaurant un état connu d'avant l'attaque sur le CPU et que cette solution est à évaluer, elle restera probablement plus lente qu'une interruption lancée juste pour arrêter un processus et non tout l'environnement du CPU. Encore une fois on se confronte entre le choix d'une sécurité plus forte au détriment des performances.

VI

Conclusion & Perspectives

Sommaire

6.1 Conclusion

70% des attaques informatiques concernent les attaques par corruption de mémoire. Ces attaques sont un fléau depuis les années 90, et nous avons vu que leur proportion n'a pas diminué depuis 2006, bien au contraire. Avec l'explosion attendue du nombre d'objets connectés, notamment par l'avènement de la 5G, la mise en réseau de dizaines de milliards d'unités fait peser des menaces à des échelles encore plus grandes. Cela concerne d'abord les noeuds terminaux, qui ont la particularité d'agir avec leur environnement. Cela s'étend à l'infrastructure réseau, avec notamment les passerelles, jusqu'au Cloud et à l'Internet. Chaque élément de cette chaîne doit être suffisamment robuste, et ce malgré des ressources fortement limitées parfois. Ces problématiques de sécurité ne sont pas nouvelles, et la littérature fait état de nombreuses approches visant à protéger ces systèmes électroniques vis-à-vis des attaques par corruption de mémoire. Mais dans le contexte de l'Internet des Objets, sont-elles suffisantes ? Toutes les études sur l'état de la sécurité dans l'Internet des Objets tendent à dire que les déploiements de sécurité ne sont pas suffisants et que nombreux sont les objets à contenir encore des vulnérabilités. De plus, les services sont mis à jour régulièrement, grâce à des mises à jour logicielles qui peuvent potentiellement introduire de nouvelles vulnérabilités. Pour mieux comprendre les limitations, nous avons mis en place une plateforme expérimentale (prototype basé sur un SoC Zynq-7000) pour étudier ces attaques et les différentes étapes d'exploitation.

Cette plateforme nous a également permis de disposer d'un environnement réaliste pour comprendre les limitations des protections existantes visant à contrer les attaques par corruption de mémoire. Nous avons proposé une classification des méthodes de protection en deux grandes catégories: les méthodes de prévention et les méthodes de tolérances aux attaques. Ces premières ont pour objectif d'éliminer un maximum de vulnérabilités d'un programme, et d'empêcher l'exécution de certaines attaques. Une multitude de méthodes existe, mais nous avons vu que chacune comporte des faiblesses rendant ces protections contournables. Les méthodes de tolérance aux attaques ont pour objectif de détecter des attaques (ou anomalies) sur le système en cours de fonctionnement, pour proposer un rétablissement du système et arrêter l'attaque en cours. Là encore, une pléthore de techniques existe. Le StackGuard, bien qu'efficace pour détecter un dépassement de tampon sur la pile, peut être contourné et ne permet pas de protéger le programme contre tous les types d'attaques. Les CFI logiciels et matériels restent intéressants, mais présentent des inconvénients non négligeables. Les approches logicielles demandent un surcoût trop élevé en performance alors que les implémentations matérielles sont difficilement déployables. De plus ces méthodes ne sont pas capables de détecter des changements malicieux sur les données. On trouve alors les solutions à base de compteurs de performance, permettant de détecter des anomalies. Cependant, une analyse plus poussée, notamment sur l'utilisation des évènements et du machine learning, était nécessaire.

Nous avons travaillé en premier lieu sur ce type de méthodes qui profitent d'une part des performances du matériel pour la récupération des données d'exécution utile à la classification et d'autre part, des possibilités induites par le logiciel pour implémenter la classification. À travers les travaux présentés dans le chapitre 4, nous avons apporté une méthodologie complète pour la mise en oeuvre de la détection. Nous avons détaillé l'approche et le raisonnement derrière la technique, le choix des évènements, et la sélection des modèles de ML pour nous approcher du meilleur ratio pénalité/précision possible. Nous avons ensuite mis en place trois cas d'usages, en démontrant le fonctionnement de la solution pour les modes de profilage *complet* et *échantillonné*. Le choix final s'est porté sur le modèle supervisé QDA, que nous avons plus amplement testé de manière empirique sur différents algorithmes de cryptographie et de tri. Ces travaux ont permis de définir une méthode de détection d'anomalies complète en améliorant drastiquement les coûts en performance par rapport aux techniques actuelles (<1% face

à plus de 10%), tout en gardant un taux de détection très élevé (jusqu'à 98.70%) et très peu de faux négatifs (<1%). Ces travaux ont aussi pointé plusieurs perspectives d'évolution et limitations comme la gestion des labels, la fréquence d'échantillonnage, ou encore la gestion des alertes (notamment des erreurs de classification).

Nous avons ensuite travaillé sur les méthodes de rétablissement, en proposant tout d'abord l'étude d'un étage de diagnostic. L'idée était d'utiliser ce dernier comme moyen de vérification des alertes de l'étage de détection pour éliminer les faux positifs et ainsi, améliorer la précision de la détection et les performances de la solution. Le postulat de départ était de mettre en place une méthode de diagnostic dont le coût en performance soit inférieur à celui de la contremesure. Nous nous sommes alors inspirés des méthodes de CFI qui suivent le flot d'exécution d'un programme en temps réel pour proposer un outil qui vérifie ce flot seulement à la demande. Même si nous n'avons pas fourni d'expérimentations complètes, nous avons proposé une première étude du fonctionnement d'un tel outil. Nous avons rencontré deux problèmes majeurs: la gestion sécurisée de la mémoire utilisée pour stocker les blocs de base empruntés, et l'incapacité de la méthode à diagnostiquer des attaques sur le flot de données, introduisant deux vulnérabilités critiques dans la méthode.

Suite à l'étude du diagnostic, nous avons évalué la combinaison de méthodes de détection et de contremesures pour déterminer si un attaquant est capable de contourner le mécanisme de rétablissement. En effet, nous avons vu que certaines méthodes de détection, qu'elles soient basées sur des anomalies, des signatures, ou des CFI, laissent une fenêtre à l'attaquant. C'est-à-dire que l'attaquant peut exécuter son attaque avant qu'elle ne soit détectée. Ces méthodes, qui peuvent être basées sur un coprocesseur, reposent alors sur une notification pour informer le CPU d'une attaque. Nous avons vu que cette notification utilise une interruption dans la plupart des cas, et que cette dernière peut finalement être contournée en exploitant la fenêtre d'attaque. Suite à cette analyse, nous avons proposé deux attaques (de type ROP) sur l'interruption, et montré que même avec une détection opérationnelle, il était possible de compromettre le système en contournant la contremesure. L'attaque demande cependant des prérequis comme des gadgets, des modes d'exécution de code, ou encore des réglages particuliers pour l'interruption.

Nous avons finalement proposé une méthode plus fiable, mais aussi plus lourde pour permettre au système de retrouver un état connu, quelles que soient les opportunités de l'attaquant. La méthode consiste à réinitialiser le système de manière logicielle dans un environnement multiprocesseur en ciblant le processeur fautif. Nous avons également étudié la capacité de l'attaquant à corrompre cette nouvelle méthode, et aucune menace n'a pu être identifiée. Ces travaux ont mis en évidence l'importance du dernier étage de sécurité qui est la contremesure. En effet, même si l'étage de détection est parfait et qu'il permet d'alerter pour toutes les attaques, une mauvaise implémentation de la contremesure peut balayer la sécurité du système. C'est un point crucial à prendre en compte lors de l'élaboration de la sécurité d'un système.

Finalement, nous avons travaillé sur tout le cycle de protection d'un système face à une famille d'attaques bien précise: les attaques par corruption de mémoire. Nous avons obtenu des compétences sur les différentes étapes de développement de solutions de sécurité, partant des méthodes de prévention, passant par les méthodes de détection, jusqu'aux méthodes de diagnostic et de contremesure. Et même si l'étude nous a permis de répondre à de nombreuses questions, elle a aussi défini d'autres problématiques.

6.2 Perspectives

Chaque partie étudiée dans le cadre de cette thèse, notamment au niveau des contributions, a soulevé de nouvelles problématiques, ou mis en évidence des améliorations à poursuivre.

Plusieurs pistes d'amélioration ont déjà été évoquées pour la méthode de détection. Par exemple, nous pourrions faire l'inférence d'un modèle QDA en C ou en VHDL pour tester d'une part les différences de performance entre une version logicielle et une version matérielle, et d'autre part l'exécution du modèle directement sur le système embarqué. Une autre idée serait d'étudier la temporalité entre les échantillons, les labels et la durée des attaques pour mieux comprendre les limitations de la solution. Ainsi, il serait possible de proposer un outil de génération de jeu de données plus performant lors de la labélisation, qui aboutirait sur une précision améliorée de la détection. Enfin des perspectives d'amélioration du traitement des erreurs et de la détection de ces dernières sont proposées à travers l'étage de diagnostic. Il propose d'améliorer la classification en traitant tous les faux positifs pour limiter les surcoûts en performance de la contremesure et la perte de contexte de l'application due à un redémarrage. De plus, nous pourrions imaginer un modèle qui génère de moins en moins de faux positifs, car si le diagnostic est capable de les détecter, alors nous pouvons affiner le modèle dans le temps.

Lors de nos derniers travaux sur la méthode de diagnostic, nous avons proposé de vérifier le flot de contrôle d'un programme à la demande. Bien que nous ayons posé les principes généraux et quelques pistes d'implémentation, il reste du travail pour avoir une implémentation complète et faire une évaluation des performances de la solution. Plusieurs pistes restent à explorer sur l'instrumentation du programme pour récupérer les informations nécessaires au diagnostic de manière sécurisée. Il restent des questions d'évaluation de performance de cette instrumentation pour que les performances du diagnostic ne soient pas supérieures à celles de la contremesure. De plus, il serait intéressant d'étudier la faisabilité de l'instrumentation sur un système d'exploitation temps réel de type FreeRTOS ou encore sans OS. Il pourrait également être intéressant d'étudier les possibilités de lancer l'instrumentation uniquement lorsque celà est nécessaire pour le diagnostic. Ainsi, nous gagnerions en performance sur l'exécution générale du programme, mais perdrions du temps sur le diagnostic et le rétablissement du système. Tous ces points sont nécessaires pour déterminer si la solution sera utilisable dans un système à ressources contraintes et par exemple, s'intégrer dans des fonctionnalités standards du LwM2M.

Enfin, durant nos travaux sur le rétablissement du système, nous avons mis en évidence une vulnérabilité sur la combinaison de méthodes de détection et de contremesures. Suite à cette découverte, nous avons proposé une méthode de réinitialisation et de redémarrage ciblant un CPU dans un environnement multiprocesseur. Bien que cette contremesure soit efficace et non altérable, elle est très contraignante pour le système; ce dernier, même si c'est partiellement, doit redémarrer et recommencer toutes ses initialisations jusqu'à un retour à la normale. Une suite possible serait alors d'améliorer cette réinitialisation pour limiter l'impact sur les performances. En effet, même si l'on peut se permettre une telle perte de performance en cas d'attaque, une si grosse perte devient désastreuse en cas de faux positif. De plus, plus court sera le retour à la normale, meilleure sera la contremesure. C'est pourquoi il serait intéressant d'étudier une méthode de sauvegarde d'état, pour être capable de rétablir cet état lors du lancement de la contremesure. L'idée serait de sauvegarder périodiquement des éléments nécessaires à ce rétablissement dans un espace sécurisé inatteignable par une application tierce. Nous pourrions ensuite utiliser la même méthode qu'actuellement pour reprendre le cours de l'exécution de l'application à la dernière sauvegarde. Un gain de performance non négligeable

serait gagné, et pourrait permettre de potentiellement se passer d'un étage de diagnostic.

Au final, la sécurité dans l'IdO se confronte aux contraintes des environnements qui proposent peu de ressources. La sécurité s'en retrouve freinée, et il est important de travailler sur des solutions de protection impactant le moins possible les performances, que ce soit à travers des utilisations ou des optimisations de fonctionnalités déjà existantes. Le développement de l'Intelligence Artificielle (IA) est une piste intéressante pour pallier aux problèmes de sécurité actuels. Nombreux sont les travaux à utiliser ces concepts, et nous avons vu une utilisation possible du ML pour détecter des anomalies lors de cette thèse. Cependant, les caractéristiques du ML peuvent être exploitées par un attaquant qui essaie de tromper la classification par des méthodes d'*Adversarial Machine Learning*. Trois stratégies principales existent. L'évasion, où l'attaquant envoie des informations non prévues au modèle pour échapper à la politique de protection. L'empoisonnement, où l'attaquant essaie de contaminer le jeu de données d'entraînement dans le but de classifier des attaques comme des comportements normaux. Et l'extraction de modèles, qui vise à étudier le modèle (boite blanche ou boite noire) pour le reconstruire et trouver de meilleurs vecteurs d'attaques. Les attaques sur le machine learning sont nombreuses, et il est important de prendre en compte tous ces cas pour avoir des modèles les plus robustes possibles. De plus l'utilisation d'informations indirectes immuables (comme les HPC) pour effectuer la classification est intéressante dans le sens où l'attaquant n'a pas un contrôle direct sur ces données. Il reste à étudier si l'attaquant est capable d'altérer ces données précisément tout en lançant son attaque. Cependant, cette dernière s'en retrouverait fortement compliquée.

A

Annexes

Programme 1

```
#include <stdio.h>
#include <stdlib.h>

#define CHAR_ARRAY_SIZE 32


int main(int argc, char * argv[])
{
    int idx = -1;
    int buffer[10] = { 0 };

    if (argc >= 2) {
        idx = atoi(argv[1]);
    }

    /* POTENTIAL FLAW: Attempt to write to an index of the array that is above th
    * This code does check to see if the array index is negative */
    if (idx >= 0)
    {
        buffer[idx] = 1;
        for(int i = 0; i < 10; i++)
        {
            printf("%d", buffer[i]);
        }
        printf("\n");
    }
    else
    {
        printf("ERROR: Array index is negative.\n");
    }

    return 0;
}
```

Programme 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CHAR_ARRAY_SIZE 32

void hijack()
{
   char ch;
   char buff[255];
   FILE *fptr;

   if ((fptr = fopen("/etc/passwd", "r")) == NULL){
       exit(1);
   }

   while (!feof(fptr)) {
       fgets(buff, 255, (FILE*)fptr);
       if (ferror(fptr)) {
           break;
       }
       printf("%s", buff);
   }

   fclose(fptr);
}


void func(int mode)
{
    char *data;
    char dataBadBuffer[50];
    char dataGoodBuffer[100];
    if (mode == 1) {
        data = dataBadBuffer;
    } else {
        data = dataGoodBuffer;
    }
    data[0] = '\0';
    {
        char source[100];
        int idx = 0;
        for(int i=0;i<100;i++) {
            source[i] = ((int)(&hijack) & (0xff<<(idx*8))) >> (idx*8);
            idx++;
            if (idx%4==0) {idx=0;}
        }
```

```
        source[100-1] = '\0';
        /* POTENTIAL FLAW: Possible buffer overflow if the size of data is less
        memcpy(data, source, 100);
        printf(data);
    }
}


int main(int argc, char * argv[])
{
    if (argc >= 2) {
        int mode = atoi(argv[1]);
        func(mode);
    }
    return 0;
}
```

Programme 3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void usage(void) {
    printf("./echo_server ip port");
}

int main(int argc, char **argv) {

    if (argc < 2) {
        usage();
        exit(0);
    }

    int sockfd, ret;
    struct sockaddr_in serverAddr;

    int newSocket;
    struct sockaddr_in newAddr;

    socklen_t addr_size;

    int nbbytes = 0;
    char bufrecv[1024], buffer[65535];

    bzero(buffer, sizeof(buffer));

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0){
        printf("[-]Error in connection.\n");
        exit(1);
    }
    printf("[+]Server Socket is created.\n");

    memset(&serverAddr, '\0', sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(atoi(argv[2]));
    serverAddr.sin_addr.s_addr = inet_addr(argv[1]);

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) < 0
        printf("setsockopt(SO_REUSEADDR) failed");

    ret = bind(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
```

```
    if(ret < 0) {
        printf("[-]Error in binding.\n");
        exit(1);
    }
    printf("[+]Bind to port %d\n", atoi(argv[2]));

    if(listen(sockfd, 10) == 0) {
        printf("[+]Listening....\n");
    } else {
        printf("[-]Error in binding.\n");
    }

    newSocket = accept(sockfd, (struct sockaddr*)&newAddr, &addr_size);
    if(newSocket < 0) {
        perror("accept");
        exit(1);
    }
    printf("Connection accepted from %s:%d\n", inet_ntoa(newAddr.sin_addr), ntoh

    while(1){
        nbbytes = recv(newSocket, bufrecv, 1024, 0);
        if (nbbytes <= 0) {
            close(newSocket);
            exit(1);
        }
        // Format string bug /!/
        sprintf(buffer, bufrecv);
        send(newSocket, buffer, strlen(buffer), 0);
        if (strncmp("/bin/sh\x00", bufrecv, 8) == 0) {

            return 0;
        }
        bzero(buffer, sizeof(buffer));
        bzero(bufrecv, sizeof(bufrecv));
    }
}
```

Programme d'exploitation du programme 3

```
from pwn import *
import sys


def main():
    r = None
    elf = None
    libc = None
    if len(sys.argv) >= 2:
        mode = sys.argv[1]
        if mode == '-r':
            if len(sys.argv) >= 3:
                host = sys.argv[2]
                port = sys.argv[3]
                r = remote(host, int(port))

        generate_traffic(r)
        exploit(r, elf, libc)


def generate_traffic(r):
    import random
    import time

    log.info("Generate traffic")

    charset = []
    for c in range(20, 255):
        if chr(c) == "%":  # remove % to not be interpreted by the format string
            continue
        charset.append(chr(c))

    num_loop = random.randint(30, 1500)

    for i in range(num_loop):
        log.info("Send random payload")
        payload = ''.join(random.choice(charset) for _ in range(random.randint(1
        r.sendline(payload)
        r.recvuntil(payload)
        time.sleep(random.randint(1, 20)/50)

    log.info("Traffic generation done")


def exploit(r, elf, libc):
    log.info("Start exploit !")

    strcmp_got = 0x0001100c + 0x00400000
    system = 0x89a + 0x00400000
```

```
    payload1 = "%{:d}x%21$hn".format((system >> 16))
    payload2 = "%{:d}x%22$hn".format(system & 0xffff - (system >> 16))

    payload = (payload1 + payload2).ljust(32, "a") + p32(strcmp_got+2) + p32(strc

    #  notify the wrapper attack starts
    d = remote("192.168.1.10", 4000)
    d.close()

    r.send(payload)
    r.close()

    #  notify the generator attack is done
    r = remote("192.168.1.100", 4000)
    r.close()


if __name__ == "__main__":
    context.log_level = "info"  # override log_level to speed up the attack
    main()
```

Bibliographie

[1] Jason Brownlee. A tour of machine learning algorithms. https://www.datasciencecentral.com/profiles/blogs/a-tour-of-machine-learning-algorithms-1?overrideMobileRedirect=1. Accessed: 2020-05-28. vi, 85, 86

[2] B. Dorsemaine, J. Gaulier, J. Wary, N. Kheir, and P. Urien. Internet of things: A definition taxonomy. In *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, pages 72–77, 2015. 2

[3] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in internet of things: Taxonomies and open challenges. *Mob. Netw. Appl.*, 24(3):796–809, June 2019. 3

[4] Fadele Ayotunde Alaba, Mazliza Othman, Ibrahim Abaker Targio Hashem, and Faiz Alotaibi. Internet of things security: A survey. *Journal of Network and Computer Applications*, 88:10 – 28, 2017. 4

[5] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *IEEE Communications Surveys Tutorials*, 21(3):2702–2733, 2019. 4

[6] Iot security foundation. http://www.iotsecurityfoundation.org. Accessed: 2020-05-28. 4

[7] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015. 4

[8] J. Granjal, E. Monteiro, and J. Sá Silva. Security for the internet of things: A survey of existing protocols and open research issues. *IEEE Communications Surveys Tutorials*, 17(3):1294–1312, 2015. 4

[9] C. Lai, R. Lu, D. Zheng, H. Li, and X. Shen. Toward secure large-scale machine-to-machine comm unications in 3gpp networks: chall enges and solutions. *IEEE Communications Magazine*, 53(12):12–19, 2015. 4

[10] Amir Moradi. Side-channel leakage through static power. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 562–579, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 4

[11] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. 4

[12] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 4

[13] J. Wurm, K. Hoang, O. Arias, A. Sadeghi, and Y. Jin. Security analysis on consumer and industrial iot devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 519–524, 2016. 4

[14] E. Bou-Harb, C. Fachkha, M. Pourzandi, M. Debbabi, and C. Assi. Communication security for smart grid distribution networks. *IEEE Communications Magazine*, 51(1):42–49, 2013. 4

[15] Jaydip Sen. A survey on wireless sensor network security. *CoRR*, abs/1011.1529, 2010. 4

[16] Hezam Akram Abdul-Ghani, Dimitri Konstantas, and Mohammed Mahyoub. A comprehensive iot attacks survey based on a building-blocked reference model. *International Journal of Advanced Computer Science and Applications*, 9(3), 2018. 5

[17] B. Bhushan, G. Sahoo, and A. K. Rai. Man-in-the-middle attack in wireless and computer networking — a review. In *2017 3rd International Conference on Advances in Computing,Communication Automation (ICACCA) (Fall)*, pages 1–6, 2017. 5

[18] M. M. Ahemd, M. A. Shah, and A. Wahid. Iot security: A layered approach for attacks defenses. In *2017 International Conference on Communication Technologies (ComTech)*, pages 104–110, 2017. 5

[19] Rodrigo Roman, Cristina Alcaraz, Javier Lopez, and Nicolas Sklavos. Key management systems for sensor networks in the context of the internet of things. *Computers Electrical Engineering*, 37(2):147 – 159, 2011. Modern Trends in Applied Security: Architectures, Implementations and Applications. 5

[20] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association. 5

[21] Cwe-121: Stack buffer overflow. https://cwe.mitre.org/data/definitions/121.html. Accessed: 2020-05-28. 5

[22] Ang Cui, Michael Costello, and Salvatore J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013. 5

[23] Vinay Sachidananda, Shachar Siboni, Asaf Shabtai, Jinghui Toh, Suhas Bhairav, and Yuval Elovici. Let the cat out of the bag: A holistic approach towards security analysis of the internet of things. In *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '17, page 3–10, New York, NY, USA, 2017. Association for Computing Machinery. 5

[24] S. Torabi, E. Bou-Harb, C. Assi, M. Galluscio, A. Boukhtouta, and M. Debbabi. Inferring, characterizing, and investigating internet-scale malicious iot device activities: A network telescope perspective. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 562–573, 2018. 5

[25] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond. 5

[26] Lwm2m specifications. http://www.openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/OMA-TS-LightweightM2M_Core-V1_1-20180710-A.pdf. Accessed: 2020-05-28. 6, 27

[27] Matt Miller. Bluehat il 2019. https://www.youtube.com/watch?v=PjbGojjnBZQ. Accessed: 2020-05-28. 7, 10, 20

[28] M. H. Miraz, M. Ali, P. S. Excell, and R. Picking. A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont). In *2015 Internet Technologies and Applications (ITA)*, pages 219–224, 2015. 10

[29] N. Perlroth S. Shane and D. E. Sanger. Security breach and spilled secrets have shaken the n.s.a. to its core. In *The New York Times*, 11 2017. 10

[30] FireEye. Smb exploited: Wannacry use of "eternalblue". https://www.fireeye.com/blog/threat-research/2017/05/smb-exploited-wannacry-use-of-eternalblue.html. Accessed: 2020-05-28. 10

[31] JONATHAN BERR. Wannacry ransomware attack losses could reach $4 billion. https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/. Accessed: 2020-05-28. 10

[32] Elf format. http://www.cs.yale.edu/homes/aspnes/pinewiki/attachments/ELF(20)format/ELF_format.pdf. Accessed: 2020-05-28. 12

[33] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security and Privacy*, 1(5):35–43, September 2003. 14

[34] David Moore, Colleen Shannon, and k claffy. Code-red: A case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, page 273–284, New York, NY, USA, 2002. Association for Computing Machinery. 14

[35] Wikipedia. Shellcode. https://en.wikipedia.org/wiki/Shellcode. Accessed: 2020-05-28. 16

[36] Tyler Bletsch. *Code-Reuse Attacks: New Frontiers and Defenses*. PhD thesis, 2011. 17

[37] Solar Designer. Libc return exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html, 1997. Accessed: 2020-05-28. 18

[38] Nergal. The advanced return-into-lib(c) exploits: Pax case study. http://www.phrack.org/issues.html?issue=58&id=4#article, 2001. Accessed: 2020-05-28. 18

[39] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), March 2012. 19

[40] Tim Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis. 20

[41] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery. 20

[42] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 15–26, New York, NY, USA, 2008. Association for Computing Machinery. 20

[43] Jonathan Salwan. Ropgadget. https://github.com/JonathanSalwan/ROPgadget, 2016. Accessed: 2020-05-28. 20

[44] Sascha Schirra. Ropper. https://github.com/sashs/Ropper, 2013. Accessed: 2020-05-28. 20

[45] E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, 2014. 20

[46] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242, 2014. 20

[47] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 30–40, New York, NY, USA, 2011. Association for Computing Machinery. 20

[48] Cwe-457: Use of uninitialized variable. https://cwe.mitre.org/data/definitions/457.html. Accessed: 2020-05-28. 21

[49] Ratul Gupta. Windows kernel exploitation tutorial part 6: Uninitialized stack variable. https://www.exploit-db.com/docs/english/44322-windows-kernel-exploitation-tutorial-part-6-uninitialized-stack-varia pdf. Accessed: 2020-05-28. 21

[50] Halvar Flake. Attacks on uninitialized local variables. https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf. Accessed: 2020-05-28. 21

[51] Cwe-843: Type confusion. https://cwe.mitre.org/data/definitions/843.html. Accessed: 2020-05-28. 21

[52] Florent Saudel. Confusion de type en c++ : la performance au détriment de la « type safety ». https://www.amossys.fr/upload/SSTIC2016-Article-confusion_type_cpp-saudel.pdf. Accessed: 2020-05-28. 22

[53] Jeong Wook Oh. Understanding type confusion vulnerabilities: Cve-2015-0336. https://www.microsoft.com/security/blog/2015/06/17/understanding-type-confusion-vulnerabilities-cve-2015-0336/. Accessed: 2020-05-28. 22

[54] Cwe-125: Out of bounds read. https://cwe.mitre.org/data/definitions/125.html. Accessed: 2020-05-28. 22

[55] Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html. Accessed: 2020-05-28. 22

[56] HacknDo. Use after free. https://beta.hackndo.com/use-after-free/. Accessed: 2020-05-28. 22

[57] Sergei Glazunov Maddie Stone. Cve-2019-13720: Use-after-free in chrome webaudio. https://googleprojectzero.blogspot.com/p/rca-cve-2019-13720.html. Accessed: 2020-05-28. 22

[58] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 414–425, New York, NY, USA, 2015. Association for Computing Machinery. 22

[59] Community. Ctf wiki pwn heap. https://ctf-wiki.github.io/ctf-wiki/pwn/readme/. Accessed: 2020-05-28. 23

[60] Shellphish. How2heap. https://github.com/shellphish/how2heap. Accessed: 2020-05-28. 23

[61] Digilent. Zybo datasheet. https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual. Accessed: 2020-05-28. 26

[62] Xilinx. Zynq-7000 datasheet. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Accessed: 2020-05-28. 26, 67, 136

[63] ARM. Arm cortex-a9. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf, 2010. Accessed: 2020-05-28. 26, 79

[64] Yocto Project. Yocto manual. https://www.yoctoproject.org/. Accessed: 2020-05-28. 26, 67

[65] European cyber week. https://www.european-cyber-week.eu/, 2018-2019. Accessed: 2020-05-28. 28

[66] ANSSI. European cybersecurity challenge. https://www.ssi.gouv.fr/agence/cybersecurite/challenge-europeen-de-cybersecurite-ecsc-2019/, 2019. Accessed: 2020-05-28. 28

[67] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie, and David Powell. Tolérance aux fautes. 33

[68] Ivo Vieira Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Bossini Tavares Moreira. An overview on the static code analysis approach in software development. 2009. 36

[69] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.*, 68(C):18–33, December 2015. 36

[70] NIST. Juliet dataset. https://samate.nist.gov/SARD/testsuite.php. Accessed: 2020-05-28. 36

[71] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. 37, 118, 121

[72] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012. 37

[73] Michal Zalewski. American fuzzing lop (afl). http://lcamtuf.coredump.cx/afl/, 2018. Accessed: 2020-05-28. 37

[74] A. Helin. Radamsa. https://github.com/aoh/radamsa, 2016. Accessed: 2020-05-28. 37

[75] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. pages 138–157, 05 2016. 37, 117, 118

[76] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 463–469, Berlin, Heidelberg, 2011. Springer-Verlag. 37, 118

[77] Pax. https://pax.grsecurity.net/, 2000. Accessed: 2020-05-28. 38

[78] ARM. In $ARMArchitectureReferenceManual$, 2009. 38

[79] Microsoft. $https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention$. In $DataExecutionPrevention$, 2004. 38

[80] PaX Team. $https://pax.grsecurity.net/docs/vmmirror.txt$. In $RELRO$, 2005. 38

[81] PaX Team. https://pax.grsecurity.net/docs/aslr.txt. In $AddressSpaceLayoutRandomization(ASLR)$, 2003. 38

[82] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 298–307, New York, NY, USA, 2004. Association for Computing Machinery. 39

[83] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 37–44, 2011. 39

[84] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in windows vista. 2008. 39

[85] Fermín J. Serna. Cve-2012-0769, the case of the perfect info leak. 2012. 39

[86] Wikipedia. Position independent code (pie). https://en.wikipedia.org/wiki/Position-independent_code. Accessed: 2020-05-28. 39

[87] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *2009 Annual Computer Security Applications Conference*, pages 60–69, 2009. 40

[88] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. 98:5–5, 01 1998. 41

[89] Gcc. https://gcc.gnu.org/, 1987. Accessed: 2020-05-28. 41

[90] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, April 2003. 41

[91] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery. 42

[92] Microsoft. Control flow guard. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2018. Accessed: 2020-06-28. 43, 128, 138

[93] Vasilis Pappas. kbouncer : Efficient and transparent rop mitigation. 2012. 44, 128

[94] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *In Symposium on Network and Distributed System Security (NDSS*, 2014. 44

[95] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 941–955, USA, 2014. USENIX Association. 44, 128

[96] PaX Team. Rap. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf. Accessed: 2020-05-28. 44

[97] J. Li, X. Tong, F. Zhang, and J. Ma. Fine-cfi: Fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security*, 13(6):1535–1550, 2018. 44

[98] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 179–194, 2016. 44

[99] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013. 44

[100] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, page 353–362, New York, NY, USA, 2011. Association for Computing Machinery. 44

[101] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 337–352, USA, 2013. USENIX Association. 44

[102] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 369–382, Washington, D.C., August 2013. USENIX Association. 44

[103] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, CCS '13, page 199–210, New York, NY, USA, 2013. Association for Computing Machinery. 44

[104] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 941–955, USA, 2014. USENIX Association. 44

[105] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 147–163, USA, 2014. USENIX Association. 44

[106] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 914–926, New York, NY, USA, 2015. Association for Computing Machinery. 44

[107] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 927–940, New York, NY, USA, 2015. Association for Computing Machinery. 44

[108] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery. 44

[109] L. Davi, D. Lehmann, A.-R Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. *USENIX Security Symposium*, pages 401–416, 01 2014. 44

[110] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. volume 8688, pages 88–108, 09 2014. 44

[111] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, page 575–589, USA, 2014. IEEE Computer Society. 44

[112] Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (cfi), 2017. 45

[113] D. Sullivan, O. Arias, L. Davi, P. Larsen, A. Sadeghi, and Y. Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016. 45, 128

[114] S. Mao and T. Wolf. Hardware support for secure processing in embedded systems. *IEEE Transactions on Computers*, 59(6):847–854, 2010. 45, 128

[115] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(12):1295–1308, 2006. 45, 128

[116] Intel. Intel control-flow enforcement technology preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017. Accessed: 2020-05-28. 45, 46, 138

[117] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Using coresight ptm to integrate cra monitoring ips in an arm-based soc. *ACM Trans. Des. Autom. Electron. Syst.*, 22(3), April 2017. 45, 128

[118] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. Hcfi: Hardware-enforced control-flow integrity. pages 38–49, 03 2016. 45, 128

[119] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Vianney Lapotre, and Guy Gogniat. ARMHEx: embedded security through hardware-enhanced information flow tracking. In *RESSI 2017 : Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*, Grenoble (Autrans), France, May 2017. 45, 128

[120] Yubin Xia, Yutao Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012. 45

[121] ARM. In *Pointer Authentication on ARMv8.3*, 2017. 46

[122] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T.R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. *24Th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 01 2015. 46, 127

[123] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *SIGARCH Comput. Archit. News*, 41(3):559–570, June 2013. 49, 50

[124] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Trans. Archit. Code Optim.*, 13(1), March 2016. 49, 50

[125] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 457–468, New York, NY, USA, 2018. Association for Computing Machinery. 50

[126] W. Shi, H. Zhou, J. Yuan, and B. Liang. Dcfi-checker: Checking kernel dynamic control flow integrity with performance monitoring counter. *China Communications*, 11(9):31–46, 2014. 52, 53

[127] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 544–551, 2015. 52, 53, 95

[128] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, page 71–76, New York, NY, USA, 2011. Association for Computing Machinery. 52, 53, 59, 95, 107

[129] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 109–129, Cham, 2014. Springer International Publishing. 53, 59

[130] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. pages 59–67, 08 2008. 64

[131] Kaadmy. C benchmark. https://github.com/kaadmy/c_cpp_benchmark. Accessed: 2020-05-28. 64

[132] PerfMon. $https://knowledge.ni.com/$. 2018. 67

[133] OProfile. $http://oprofile.sourceforge.net/$. 2018. 67

[134] Perf Tool. $http://lacasa.uah.edu/$. 2018. 67

[135] Intel V-Tune. $https://software.intel.com/en-us/vtune-amplifier-cookbook$. 2018. 67

[136] Performance application programming interface. In $http://icl.cs.utk.edu/papi/$, 2018. 67, 68

[137] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 67, 75, 87

[138] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 18, USA, 2004. USENIX Association. 117

[139] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 18, USA, 2004. USENIX Association. 117

[140] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. 05 2012. 117

[141] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 423–427, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 117

[142] J. Troger and C. Cifuentes. Analysis of virtual method invocation for binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 65–74, 2002. 117

[143] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54, 2002. 117

[144] Avrora. http://compilers.cs.ucla.edu/avrora/cfg.html. Accessed: 2020-05-28. 118

[145] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis amp; transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. 118

[146] Hex-Rays. The interactive disassembler. https://www.hex-rays.com/products/ida/. Accessed: 2020-05-28. 118

[147] NSA. Ghidra. https://ghidra-sre.org/. Accessed: 2020-05-28. 118

[148] Radare2. https://book.rada.re/index.html. Accessed: 2020-05-28. 118

[149] Thomas Dullien and Sebastian Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. 2009. 118

[150] Valgrind developers. libvex. https://github.com/lifting-bits/libvex. Accessed: 2020-05-28. 118

[151] Anton Kochkov. Esil - universal il. https://fr.slideshare.net/AntonKochkov/slidesen. Accessed: 2020-05-28. 118

[152] Dmytro Oleksiuk. Openreil. https://github.com/Cr4sh/openreil. Accessed: 2020-05-28. 118

[153] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association. 121

[154] Unicorn engine - multi-arch multi-platform cpu emulator framework. https://github.com/unicorn-engine/unicorn. Accessed: 2020-05-28. 121

[155] Program trace macrocell. https://developer.arm.com/documentation/ihi0035/b/Introduction/About-the-Program-Trace-Macrocell. Accessed: 2020-05-28. 124

[156] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek. Integration of rop/jop monitoring ips in an arm-based soc. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 331–336, 2016. 128

[157] Mehryar Rahmatian, Hessam Kooti, Ian G. Harris, and Elaheh Bozorgzadeh. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embed. Syst. Lett.*, 4(4):94–97, December 2012. 128

[158] Andrew N. Sloss. Interrupt handling. http://classweb.ece.umd.edu/enee447.S2016/ARM-Documentation/ARM-Interrupts-1.pdf. Accessed: 2020-05-28. 131

[159] ARM. Trustzone. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed: 2020-05-28. 132

[160] E. M. Benhani, C. Marchand, A. Aubert, and L. Bossuet. On the security evaluation of the arm trustzone extension in a heterogeneous soc. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 108–113, 2017. 132, 137

[161] ARM. Axi specifications. http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf, 2003. Accessed: 2020-05-28. 143

Abstract

70% of computer attacks are memory corruption attacks. With the exponential growth of connected devices through the Internet of Things (IoT), the question of vulnerability of these embedded electronic systems is raised: what is the real threat? What solutions are available in the state of the art, and are they sufficient? How to propose relevant and functional protection solutions in an ecosystem, such as the IoT, composed of constrained environments? Initially, we study memory corruption attacks by implementing an experimental platform (prototype based on a Zynq-7000 SoC) to understand how they work, deploy existing countermeasures, and analyze their limitations. We mainly show that existing CFI solutions have some significant drawbacks: a substantial impact on performance in their software approach, an additional silicon cost in their hardware implementation, and an inability to detect attacks on data flow usage. A possible alternative is the indirect detection of attack signatures or operating anomalies through performance counters, a technique relying on the advantage of microprocessors' hardware. The literature includes many studies in this field, which we analyze to specify the basis of a relevant solution in the choice of events to predict an attack. Thus, we compare classification models derived from machine learning, particularly their accuracy and memory/-performance impact. Our analysis leads us to a method likely to generate a minimum of false negatives and sufficiently light on the resource constraints specific to our embedded domain. Thus, through an experimental approach, we can propose a solution based on the use of QDA to obtain an attack detection accuracy of up to 98.78%, with performance penalties less than 1%. For this solution to be relevant in an industrial context, it is also necessary to consider the potentially crippling number of false positives. Then, we propose a diagnostic stage following the detection stage, based on the use of a partial CFI dedicated to verifying previously identified false positives. We also show that state of the art hardware implementations of detection tools (e.g. Control Flow Integrity Check (CFI) solutions) rely on using a two-step countermeasure, which can turn out to be weak. The first part, which consists of notifying an attack to the processor through an interruption, can be bypassed before ignoring the solution's entire security policy. To solve this issue, we suggest to implement a robust reset and reboot method adapted to multiprocessor systems. Thus, in this thesis, we contribute to the security of embedded systems against some of the most dangerous computer attacks, demonstrated experimentally in an approach that meets the constraints of resources specific to the field.

Résumé

Les attaques par corruption de mémoire représentent 70% des attaques informatiques. Avec la croissance exponentielle d'objets programmables et connectés à travers notamment l'Internet des Objets (IdO), il se pose la question de la vulnérabilité de ces systèmes électroniques embarqués: quelle est la menace réelle, quelles solutions trouve-t-on dans l'état de l'art et sont-elles suffisantes? Comment proposer des solutions de protection pertinentes et fonctionnelles dans un écosystème comme l'IdO composé d'environnements contraints ? Dans un premier temps, nous étudions par la mise en place d'une plateforme expérimentale (prototype basé sur un SoC Zynq-7000) les attaques par corruption de mémoire pour comprendre leur fonctionnement, déployer les contremesures existantes et analyser leurs limitations. Nous montrons notamment que les solutions de CFI existantes posent un certain nombre d'inconvénients majeurs: un impact fort sur les performances dans leur approche logicielle, un coût silicium additionnel dans leur implémentation matérielle, et une incapacité à détecter des attaques sur l'utilisation de données. Une alternative possible consiste en la détection indirecte de signature d'attaque ou d'anomalie de fonctionnement par l'utilisation de compteurs de performance, technique qui présente l'avantage de reposer sur un matériel généralement disponible dans les microprocesseurs. La littérature compte un certain nombre d'études dans ce domaine, que nous analysons pour spécifier les bases d'une solution qui s'avèrerait pertinente dans le choix des évènements pour prédire l'occurrence d'une attaque. Nous comparons ainsi un certain nombre de modèles de classification issus du machine learning, notamment leur précision et leur impact mémoire/performance. Notre analyse du contexte applicatif nous oriente vers une méthode susceptible de générer un minimum de faux négatifs, et suffisamment légère quant aux contraintes de ressources spécifiques à notre domaine de l'embarqué. Ainsi nous arrivons, par une approche expérimentale, à proposer une solution reposant sur l'utilisation de QDA et à obtenir une précision de détection d'attaque allant jusqu'à 98.78%, avec des pénalités en performance inférieures à 1%. Pour que cette solution puisse être pertinente dans un contexte industriel, il faut aussi tenir compte du nombre potentiellement rédhibitoire de faux positifs. C'est dans cette optique que nous proposons un étage de diagnostic successif à celui de détection, s'appuyant sur l'utilisation d'un CFI partiel dédié à la vérification plus formelle de faux positifs identifiés au préalable. Nous montrons également que des implémentations d'outils de détection issues de l'état de l'art (par exemple des solutions de vérification d'intégrité du flot de contrôle (CFI)) reposent sur l'utilisation d'une contremesure, faite en deux temps, qui peut se révéler faible. La première partie consistant à notifier le processeur d'une attaque à travers une interruption peut être contournée, pour finalement ignorer toute la politique de sécurité proposée par la solution. Nous proposons d'y remédier par une méthode de réinitialisation et redémarrage robuste adaptée à des systèmes multiprocesseurs. Ainsi, dans cette thèse, nous apportons une contribution à la sécurisation des systèmes embarqués vis-à-vis des attaques informatiques parmi les plus délétères, démontrée de manière expérimentale, et ce dans une approche répondant aux contraintes de ressources spécifiques au domaine.