Vérification externalisée du flot de contrôle: Comment adapter la sécurité système à l'informatique embarquée

Valentin Lefils

Université de Lille

Thèse de doctorat

Vérification externalisée du flot de contrôle: Comment adapter la sécurité système à l'informatique embarquée

---

Auteur :
Valentin Lefils
Superviseurs :
Professeur Gilles Grimaud
Maître de conférence HDR Julien Cartigny
Rapporteurs :
Professeur Pierre Paradinas
Professeure Associée Vania Marangozova

Une thèse soumise pour l'obtention
du grade de Docteur de l'Université de Lille

au sein de

Equipe 2xs
Laboratoire CRIStAL

2 avril 2019

« *Standards are always out of date. That's what makes them standards.* »

Alan Bennett

Résumé

Les systèmes embarqués sont utilisés pour accomplir des tâches critiques dans les systèmes industriels, l'automobile ou l'aéronautique. Pourtant, beaucoup d'entre eux possèdent une faible sécurité matérielle et logicielle. Par exemple, de nombreuses attaques (dépassement de tampon, injection de faute, etc. . .) visent à modifier le comportement du logiciel embarqué et mener à une exécution de code arbitraire. Une solution à ce problème est la mise en place d'une politique de vérification de l'intégrité du flot de contrôle (Control Flow Integrity ou CFI) qui vérifie en temps réel le comportement du programme par rapport à l'exécution attendue. Mais une telle solution ne correspond pas en l'état aux contraintes de l'informatique embarquée car elle est coûteuse en terme de temps de calcul.

Dans la première partie de cette thèse j'évalue la pertinence d'une solution externalisée de CFI pour les systèmes embarqués[62]. Celle-ci repose sur l'instrumentation du code source à protéger afin de produire, lors de l'exécution, une trace envoyée à un moniteur externe qui vérifie si elle est en accord avec un graphe de flot de contrôle extrait à la compilation. J'ai évalué la faisabilité de la démarche en démontrant qu'elle était pertinente vis à vis des systèmes embarqués.

La deuxième partie de mes travaux concerne la vérification du flot de contrôle. Les solutions de CFI classiques reposent sur une analyse du code source afin de créer un modèle de référence pour la vérification de l'exécution. Mais de nombreuses études ont démontré les faiblesses de ce modèle car il est difficile de prédire avec précision le comportement du programme. Pour résoudre ce problème, j'ai proposé une solution basée sur l'apprentissage automatique[63] afin de passer d'un modèle de prédiction du comportement à un modèle de déduction du comportement. Cette solution repose sur la mise en place d'un algorithme d'apprentissage capable d'induire un modèle précis à partir de traces réelles d'exécution. Afin de valider cette démarche, l'efficacité du mécanisme d'apprentissage ainsi que la validité et la précision du nouveau modèle sont évalués. Finalement, ces travaux incluent le résultat des expériences menées sur cette nouvelle approche ainsi que des pistes de recherche pour des travaux futurs.

Embedded systems are often used to accomplish critical tasks in industrial systems such as automobile or aeronautics. Yet, many of them offer a poor hardware and software security. For example, many attacks (buffer overflow, fault injection, etc...) aim to modify software's behavior in order to lead to an arbitrary code execution. To tackle this, a control flow integrity (CFI) policy can be deployed in order to verify in real time a program's behavior. But, as it stands, this solution doesn't match with embedded systems constraints due to an important overhead implied on the target.

In the first part of this thesis, I present an externalized CFI policy for embedded systems. The target code in instrumented to produce a trace during its execution and to send it to an external monitor which verifies if the trace is correct according to a control flow graph extracted by static analysis at compilation time. I evaluated the feasibility of this solution by demonstrating that it is compatible with embedded systems constraints.

The second part of this work is about control flow integrity. Usual CFI policies imply source code analysis in order to create a reference model used to verify execution. Yet, many studies showed the weaknesses of these approach due to many imprecisions in the program's behavior modelization. In response, i present a solution based on machine learning to deduce a program's behavior from observed trusted executions. This solution use Alergia algorithm to induce a precise model from execution traces. To validate this approach, the new model's precision is evaluated and new leads for futures researches are presented.

Remerciements

Les travaux présentés dans ce document n'ont certes pas étés un long fleuve tranquille mais ils ont aboutis à un résultat que je regarde aujourd'hui avec fierté. Pour cela je remercie toutes les personnes qui m'ont aidé durant ces trois grosses années. Plus particulièrement je remercie Gilles Grimaud et Julien Cartigny, tout d'abord pour la confiance qu'ils ont placé en moi lorsqu'ils m'ont recruté pour ce projet mais également pour leur sagesse qui m'a guidé tout au long de la thèse (parfois à coup de "débrouilles toi !", mais c'est comme cela qu'on apprend). Je remercie également Louis Rilling, Frederic Majorczyk et la DGA pour leur soutien, leur encadrement et leurs conseils qui m'ont aidé à m'orienter dans ce vaste sujet. Je remercie Aurélien Lemay et Joachim Niehren qui m'ont apporté leur expertise technique sur les questions d'apprentissage automatique et d'induction de modèle, sans eux nous n'aurions jamais réussi à aboutir à un tel résultat. Je remercie mes collègues de l'équipe 2xs et tout particulièrement mes partenaires de galère : Quentin, Chris, Nadir, Mahidine et Narjes avec qui j'ai partagé les joies et les peines de la vie de thésard. Et finalement je remercie mes proches qui m'ont soutenu et supporté durant toutes ces épreuves, vous m'avez donné la force d'aller jusqu'au bout de cette aventure.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

CFI	**C**ontrol **F**low **I**ntegrity
CPI	**C**ode **P**ointer **I**ntegrity
CFG	**C**ontrol **F**low **G**raph
EE-CFI	**E**mbedded **E**xternalized **C**ontrol **F**low **I**ntegrity
CPU	**C**entral **P**rocessing **U**nit
RAM	**R**andom **A**ccess **M**emory
ROM	**R**ead **O**nly **M**emory
DMA	**D**irect **M**emory **A**ccess
SQL	**S**tructured **Q**uery **L**anguage
MSDL	**M**icrosoft **S**ecurity **D**evelopment **L**ifecycle
LLVM	**L**ow **L**evel **V**irtual **M**achine
LLVM-IR0	**LLVM I**ntermediaite **R**epresentation

A tous ceux qui m'ont soutenu et qui ont cru en moi. . .

Chapitre 1

Introduction

1.1 Contexte de la thèse

Les travaux présentés dans ce document sont le fruit de la thèse que j'ai réalisé en partenariat avec la Direction Générale de l'Armement (DGA). La DGA est une direction du ministère de l'armement et ses missions sont organisées selon 3 axes : équiper les forces armées, préparer l'avenir en développant et évaluant les nouvelles technologies et enfin promouvoir les exportations d'armement. C'est dans le cadre de la deuxième mission que viennent s'inscrire mes travaux, à travers un financement de la Mission pour la recherche et l'innovation scientifique (MRIS) sur la sécurité des systèmes informatiques.

Lors de ma thèse j'ai été encadré administrativement par le Centre National de la Recherche Scientifique (CNRS) partenaire de la DGA dans le financement des thèses. J'ai effectué mes travaux à l'Institut de Recherche sur les Composants logiciels et matériels pour l'Information et la Communication Avancée (IRCICA), institut interdisciplinaire partenaire de l'Université de Lille et du CNRS visant à développer les technologies de l'information et de la communication responsables. Dans ces locaux j'ai été accueilli au sein du laboratoire CRIStAL qui regroupe la recherche en informatique, signal et automatique à l'Université de Lille. Finalement j'ai intégré l'équipe 2XS (*eXtra Small, eXtra Safe*) dirigée par Gilles Grimaud et spécialisée dans la sécurité, la fiabilité et l'efficacité des systèmes embarqués fortement contraints.

J'ai effectué ces travaux sous la supervision du professeur Gilles Grimaud et de Julien Cartigny, maître de conférence. J'ai également pu compter sur un encadrement technique de la DGA en la personne de Louis Rilling et Frédéric Majorczyk qui ont évalué et guidé mon travail tout au long de cette thèse. Enfin, j'ai reçu l'aide de Aurélien Lemay et Joachim Niehren de l'équipe LINKS du laboratoire CRIStAL qui m'ont apporté leur expertise sur les questions touchant à l'apprentissage automatique et l'induction d'automates.

1.2 Contexte scientifique

Ces travaux s'inscrivent dans le contexte général de la sécurité des systèmes embarqués. Aujourd'hui, les systèmes embarqués font partie intégrante de nos vies. Depuis notre smartphone jusque dans notre voiture en passant par nos maisons et nos infrastructures, ces dernières années ont vu proliférer ces milliards de petits systèmes informatiques. Les dernières estimations publiées par Gartner [42] prédisent que d'ici à 2020, 20 milliards de systèmes embarqués plus ou moins évolués s'immisceront dans les moindres recoins de notre quotidien. Mais ces objets, à qui l'on confie de plus en plus de chose, sont-ils sûrs ? En 2009 Nicolas Ruff commençait sa présentation à la conférence SSTIC par ces mots : "la sécurité est un échec" et il posait le constat que réalité économique et sécurité informatique étaient difficilement conciliables. Afin d'être compétitif il faut en effet sortir le produit rapidement et à moindre coût et ce constat est d'autant plus valable pour les systèmes embarqués.

26 milliards d'objets connectés d'ici 2020, pour une population estimée alors à 8 milliards d'êtres humains, cela représente 4.2 systèmes informatique miniatures par être humain. Pourtant en 2014 déjà, la société HP affirmait que 70% des systèmes embarqués comportaient au moins une faille de sécurité [51], soit 14 milliards de dispositifs vulnérables. Entre la conception, le développement et la mise en production, chaque étape est susceptible d'introduire ses failles de sécurité. Entre 2010 et 2012 la société TRENDnet a commercialisé des caméras de surveillance dont les identifiants étaient transmis en clair sur le réseau [100], rendant ces dispositifs accessibles à toute personne mal intentionnée. En 2015 une équipe de chercheurs réussit à prendre le contrôle d'un SUV de la marque JEEP grâce au réseau cellulaire[11]. En 2010 le vers Stuxnet [96] est soupçonné d'avoir détruit un millier de centrifugeuses à uranium en Iran avant d'être neutralisé. 2016, Carson Block publie un rapport sur les pacemakers distribués aux patients par l'hôpital St Jude de Memphis [93], ceux-ci révèlent une vulnérabilité permettant à l'attaquant d'en prendre le contrôle permettant d'éteindre le dispositif ou de délivrer une décharge pouvant être létale. 2016 toujours, le réseau de bots informatiques ou *botnet* Mirai [8], composé de centaines de milliers de dispositifs embarqués compromis, lance une attaque par déni de service sur l'hébergeur français OVH qui a relevé un pic d'attaque à 1Tbits par seconde. Et cette liste, qui n'est absolument pas exhaustive, continue sans cesse de s'allonger.

Alors quelles sont les causes de ce nombre important de vulnérabilités ? Premièrement, comme le disait Nicolas Ruff elles sont, comme dans le reste des systèmes informatiques, en parties dues à la réalité économique. Afin d'optimiser le profit, les coûts de développement sont réduits au strict minimum, les délais de développement raccourcis au maximum et la production éparpillée aux quatre coins du monde. Mais ça ne suffit pas à expliquer de tels chiffres, les systèmes embarqués souffrent de contraintes supplémentaires qui impactent moins les systèmes traditionnels. Tout d'abord ces systèmes reposent sur un cycle de développement long. Entre la conception du dispositif, le développement du code, la mise en production et le déploiement, il peut s'écouler jusqu'à deux ans. De plus, ces systèmes sont le plus souvent impossibles à mettre à jour. Ils sont fournis avec un logiciel intégré et il faut renouveler le matériel afin de proposer une nouvelle version du logiciel. Cet état de fait implique que si une vulnérabilité est détectée après la mise en production, alors il faudra attendre plusieurs mois voir des années pour que le code soit patché, le dispositif mis en production et redéployé. La plupart du temps le dispositif vulnérable n'est même pas remplacé pour des raisons économiques. De plus les systèmes embarqués étant peu puissants, ils offrent peu

de mécanismes de sécurité matériel et logiciel et la moindre faille de développement devient alors facilement exploitable.

1.3 Cadre de la thèse

Ces travaux s'intéressent à la sécurité des systèmes embarqués et plus précisément à l'Internet des Objets (*Internet of Things* ou IoT). Dans cette étude ces dispositifs sont définis par une puissance de calcul réduite (inférieur à 1Ghz) et par leur interconnexion. Nous excluons donc les dispositifs plus puissants tels que les smartphones qui reposent sur un matériel et des logiciels qui se rapprochent des systèmes classiques en terme de mécanismes de sécurité. De plus nous excluons de cette étude les cartes à puce de type *Java Card* car celles-ci reposent sur une architecture différente des autres systèmes, utilisant une machine virtuelle Java et proposent leurs propres mécanismes de sécurité.

Cette étude vise à proposer une sécurisation de ces systèmes contre l'exploitation des failles de développement. Celles-ci ne modifient pas le comportement attendu du programme dans un contexte d'utilisation normale mais elles peuvent permettre à l'attaquant de dérouter le programme en envoyant des entrées volontairement mal formées. L'exploitation de failles de conception (transmission en clair des identifiants, *backdoor*, identifiants par défaut, etc...) n'implique pas un détournement du comportement normal du programme, ces failles ne peuvent donc être réparées que par des bonnes pratiques de conception. De plus nous excluons également de cette étude toutes les méthodes de prévention d'intrusion comme l'analyse des paquets réseau et donc toutes les solutions de type pare-feu. Ces travaux s'intéressent au méthodes de détection d'intrusion, notre objectif étant de pouvoir déterminer si un dispositif embarqué a été compromis. De plus, cette étude a pour vocation de s'insérer dans le processus de développement existant sans impliquer un changement de matériel ou de logiciel afin de correspondre à un maximum de systèmes embarqués et de s'intégrer facilement au processus de développement des systèmes industriels. Pour faire cela nous proposerons une solution basée uniquement sur le renforcement automatisé du logiciel existant lors de la phase de compilation.

1.4 Objectifs de la thèse

Les travaux présentés dans ce document vont tenter de résoudre le problème de la détection de la compromission d'un système embarqué dédié à l'internet des objets. Cette détection prendra la forme d'une vérification de l'exécution du code exécuté par ces dispositifs afin de détecter une différence entre le comportement observé et le comportement attendu provoqué par l'exploitation d'une faille de conception. Après avoir présenté plus en détail la problématique, nous verrons dans un premier temps comment une externalisation de la surveillance permet d'adapter des solutions de sécurité fortes à des systèmes embarqués contraints. Nous verrons que cette solution permet de surveiller tout ou partie du code présent sur le dispositif embarqué en proposant comme exemple la surveillance du système d'exploitation temps réel FreeRTOS.

Ensuite dans un second temps nous étudierons la couverture de sécurité des solutions actuelles de CFI. Nous verrons ainsi comment les branchements indirects et la présence de code non utilisé induisent une sur-évaluation des exécutions possibles menant à un

laxisme de la solution de sécurité. En réponse à cette problématique nous proposerons une solution basée sur l'apprentissage automatique visant à cerner au mieux le rôle des appareils embarqués afin d'en identifier un changement de comportement qui n'aurait pas été détecté par les systèmes de sécurité classiques.

1.5 Structure du document

Ce document de thèse s'articule autour de 6 chapitres, incluant ce chapitre d'introduction.

Le **Chapitre 2** présente un état de l'art sur l'attaque et la défense du flot de contrôle d'un programme. Il détaille également l'état de l'art de ce sujet dans le contexte des systèmes embarqués. Finalement ce chapitre décrit comment les mécanismes de défense peuvent s'intégrer dans le processus de développement d'un système.

Le **Chapitre 3** présente en détail la problématique à laquelle nous proposons de répondre dans ce document. Cette problématique est composée de deux axes : comment adapter la sécurité du flot de contrôle à l'informatique embarquée et comment utiliser les spécificités de ces systèmes afin de répondre aux problématiques de la défense telle qu'elle est proposée actuellement. Ce chapitre détaille également la proposition qui a été faite pour chacune de ces problématiques.

Le **Chapitre 4** présente notre réponse à la première problématique, à savoir une solution d'externalisation de la solution de sécurité. L'objectif de ce chapitre est de montrer la pertinence de cette démarche en démontrant que celle-ci est adaptée aux contraintes des systèmes embarqués.

Le **Chapitre 5** présente notre réponse à la deuxième problématique. Nous démontrerons comment l'apprentissage automatique à partir de traces d'exécutions saines permet la construction d'un nouveau modèle de référence pour la sécurité. Ce chapitre présente une évaluation des mécanismes d'apprentissage automatique ainsi qu'une évaluation de sécurité du nouveau modèle.

Le **Chapitre 6** conclut ces travaux en résumant les principales contributions et en montrant en quoi celles-ci répondent à l'état de l'art sur la sécurité du flot de contrôle dans les systèmes embarqués. Finalement dans ce chapitre je reviendrais sur les choix que j'ai fait durant cette thèse et sur les perspectives ouvertes par celle-ci.

Chapitre 2

Etat de l'art

Comme il a été pointé dans l'introduction, la sécurité des systèmes embarqués est un enjeu majeur car ils possèdent peu de mécanismes matériels et logiciels de protection, malgré le fait que les problèmes de sécurité dont ils sont victimes sont connus depuis de nombreuses années. Les systèmes embarqués n'étant pas équipés des dernières solutions de sécurité par soucis de réduction des coûts, ils souffrent encore de certains problèmes historiques qui sont aujourd'hui réglés dans les systèmes classiques. Dans cette section nous allons faire un tour d'horizon de l'évolution des techniques de sécurité qui entrent en jeu lors de l'exécution du programme, puis nous ferons un parallèle avec les caractéristiques que l'on retrouve aujourd'hui dans les systèmes embarqués.

2.1 Introduction à l'exécution de programmes sur un micro-processeur à pile (architecture von neumann)

Lorsqu'un programme est exécuté par des systèmes équipés de micro-processeurs (pc, téléphone, micro-contrôleurs embarqués, carte à puce, etc. . .) son exécution est régie à la fois par le matériel ainsi que par le logiciel. Plus exactement, le processeur exécute des séries d'opérations élémentaires nommées instructions afin d'effectuer un ensemble de calculs sur des données. De nos jours, l'immense majorité des systèmes informatiques sur le marché sont des ordinateurs à programme enregistrés, ce qui signifie qu'ils enregistrent les instructions des programmes dans lur mémoire vive. Parmi les ordinateurs à programme enregistrés il faut distinguer deux grandes familles : les architectures Von Neumann qui enregistrent les données et les instructions dans une mémoire unique et les architectures Harvard qui utilisent des mémoires différentes pour stocker données et instructions. Dans ce document nous nous intéresserons à la sécurité des systèmes reposant sur une architecture Von Neumann.

2.1.1 Détail de l'architecture Von Neumann

La figure 2.1 présente l'architecture Von Neumann, on peut la décomposer en quatre entités principales :

La mémoire chargée du stockage des données.

L'unité de contrôle (UC) dont le rôle est de récupérer les instructions en mémoire

L'unité arithmétique et logique (UAL) effectue l'opération à partir de l'instruction fournie par l'UC et potentiellement des données présentes dans la mémoire.

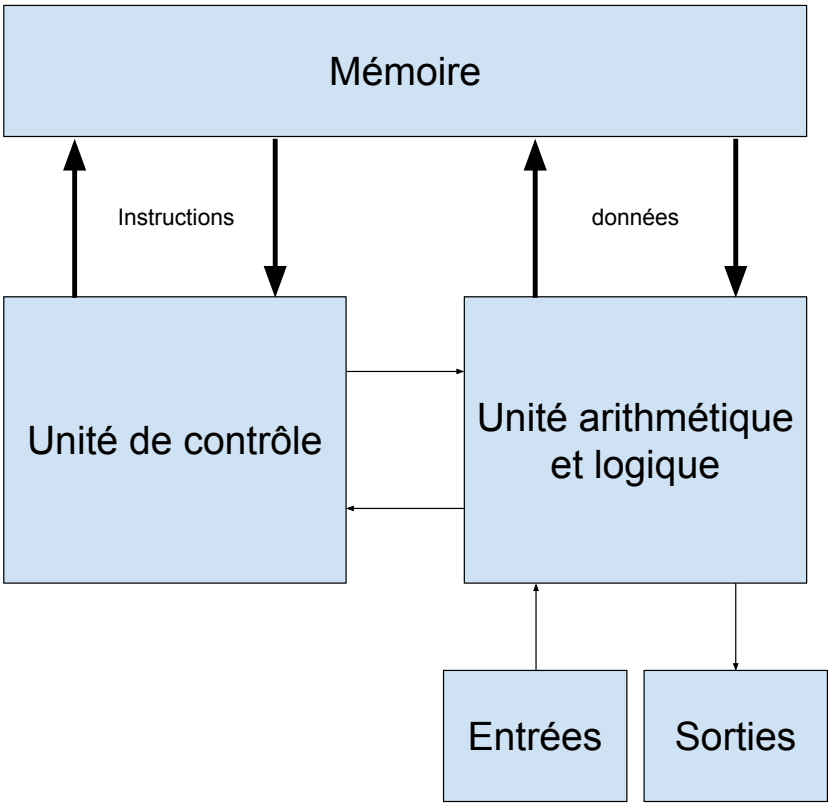

FIGURE 2.1 – Schéma général du modèle de Von Neumann

Les entrées/sorties représentent quant à elle la communication avec le monde extérieur (autre matériel).

L'architecture Von Neumann repose donc sur l'utilisation d'une mémoire unique pour le stockage des instructions et des données, cette mémoire va donc être manipulée à la fois par l'UC et par l'UAL. Dans la réalité du matériel, cette mémoire prend la forme de registres intégrés au microprocesseur qui servent de mémoire temporaire. Cependant, si ces registres offrent une rapidité d'accès, leur capacité est limitée (16 x 64 bits sur un processeur 64 bits), de fait il est nécessaire d'avoir également une mémoire moins directe mais plus conséquente, la mémoire vive, afin de stocker un programme complexe et ses données. C'est donc notamment la manipulation de cette mémoire que l'on va retrouver dans les entrées et sorties.

2.1.2 Code machine

Le code binaire (ou code machine) du programme est la représentation du programme sous la forme d'une suite d'instructions. Il est stocké dans certains cas dans la mémoire persistante exécutable (*Read Only Memory* ou ROM), celle-ci est en lecture seule et donc a la propriété de ne pas être modifiable lors de l'exécution. Cependant, dans la plupart des cas il est stocké dans de la mémoire modifiable : la mémoire persistante de stockage (disque dur, mémoire flash). Dans ce cas le programme est d'abord chargé dans la mémoire vive avant d'être exécuté, cette mémoire permet un accès direct à chaque zone mémoire contrairement à la mémoire persistante qui offre un accès séquentiel et donc moins rapide. Une fois ce processus réalisé nous parlerons alors de segment de code comme étant la représentation d'un programme en mémoire exécutable. Le processeur exécute ensuite les instructions dans l'ordre, en parcourant le segment de code. Afin de savoir l'instruction suivante à exécuter, le processeur garde dans un registre un compteur appelé compteur ordinal ou pointeur d'instruction pointant vers la zone mémoire correspondant à la prochaine instruction à exécuter.

Cependant l'exécution d'un programme n'est en général pas linéaire, un programme se compose de branchements conditionnels, de boucles et de fonctions, dans le but de rendre son écriture moins fastidieuse. Dans ce cas des instructions spécifiques modifient la valeur du compteur ordinal pour sauter vers une autre adresse de code, comme par exemple revenir quelques instructions en arrière afin de réaliser une boucle. Dans le cas des fonctions, il s'agit d'insérer artificiellement une série d'instructions au milieu d'une autre. Une instruction spécifique nommée "appel" vient notifier qu'il faut désormais exécuter une autre portion de code et, une fois cette portion de code exécutée (la fin étant signalée par une instruction "retour"), le programme reprend le cours de son exécution. Pour cela il faut que l'instruction d'appel désigne une adresse en mémoire pointant vers le code de la fonction. Il faut également enregistrer la valeur du compteur ordinal avant l'appel afin de pouvoir y revenir au moment de l'instruction de retour. Cette valeur est appelée adresse de retour. L'ensemble des opérations visant à interagir avec le compteur ordinal est appelé flot d'exécution ou flot de contrôle du programme, concrètement le flot d'exécution représente l'ordre dans lequel les instructions d'un programme sont exécutées.

2.1.3 Contexte d'exécution

Un programme est donc composé d'une suite d'instructions et de données qui évolueront au fil de l'exécution du programme : résultats intermédiaires, compteurs ou

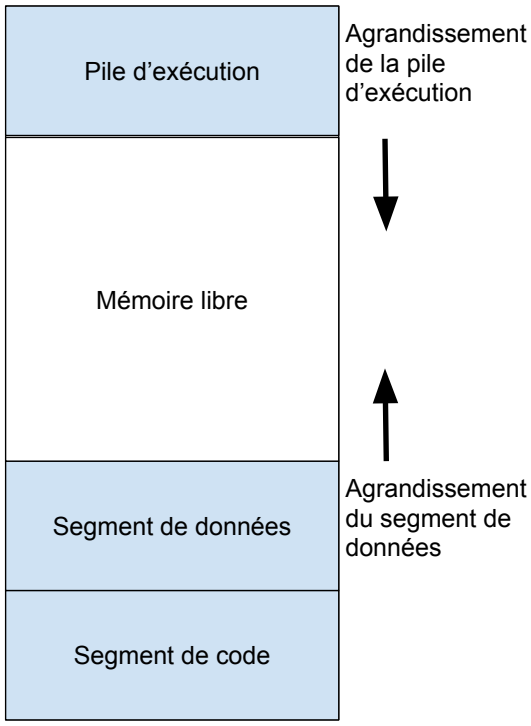

FIGURE 2.2 – Vue d'ensemble de la mémoire lors de l'exécution d'un programme

données externes. Ces données sont divisées en deux catégories : les données statiques et les données dynamiques.

Données statiques : Ces données ont une durée de vie égale à celle du programme, un espace mémoire leur est alloué au lancement du programme et il ne sera libéré qu'a la terminaison de celui-ci. Ces données sont sauvegardées dans le segment de données.

Données dynamiques : Ces données ne sont accessibles que depuis une portion du programme comme par exemple dans le contexte d'une fonction. Dans ce cas, la mémoire n'est allouée que lors de l'entrée dans la fonction puis libérée à sa sortie, la donnée est alors détruite. Ces données sont sauvegardées sur la pile d'exécution.

Lorsqu'un programme est factorisé en fonctions, chaque fonction possède sa propre sous-partie de la pile d'exécution. La figure 2.2 représente une de ces sous partie, celle-ci est composée des données fournies à la fonction lors de l'appel, des données intermédiaires propres à la fonction et également de l'adresse de retour de la fonction. L'ensemble de ces données est appelé contexte d'exécution de la fonction.

2.1.4 Pour résumer

Sur une architecture de Von Neumann, un programme est donc une série d'instructions enregistrées en mémoire qui manipule des données enregistrées dans cette même

mémoire. Les instructions sont chargées par l'unité de contrôle qui dirige ensuite l'unité arithmétique et logique pour effectuer les calculs nécessaires en lisant et écrivant la mémoire selon l'instruction.

2.2 Gestion de la mémoire, privilèges et systèmes d'exploitation

Si l'exécution d'un programme sur une architecture théorique de Von Neumann semble relativement simple, dans les systèmes actuels de nombreux mécanismes sont mis en place afin d'apporter au système des garanties supplémentaires. Parmi ces mécanismes, il est nécessaire pour la suite de détailler le système de gestion de la mémoire et de privilège qui est à la base des noyaux de système d'exploitation (*Operating system* en anglais ou OS).

Cette section ne se veut pas exhaustive dans sa définition d'un système d'exploitation et de son noyau, car cette étude concerne des systèmes embarqués n'offrant pas toujours de mécanismes de gestion mémoire et les solutions que nous proposons ne reposent donc pas sur les spécificités des OS et de leur noyau. Cependant, afin de bien comprendre certaines solutions de l'état de l'art il est nécessaire de définir quelques notions rudimentaires sur la mémoire et les privilèges.

2.2.1 MMU / MPU

Dans le cadre d'une architecture Von Neumann, la mémoire est accessible à toutes les instructions directement via un système d'adressage physique de la mémoire, de ce fait, tous les programmes ont donc accès à l'intégralité de la mémoire. Cependant pour des raisons de sécurité, un partitionnement de la mémoire en plusieurs zones ayant des restrictions d'accès différentes peut être nécessaire.

Pour réaliser un partitionnement de la mémoire il est nécessaire d'introduire un nouveau composant : l'unité de protection mémoire (*Memory Protection Unit* en anglais ou MPU). La MPU se comporte comme une interface entre le programme et la mémoire en autorisant ou refusant l'accès à certaine zone en fonction des droits attribués au programme.

Aujourd'hui, certains systèmes intègrent un composant plus évolué que la MPU appelé unité de gestion de la mémoire (*Memory Managment Unit* en anglais ou MMU). La MMU fournit une interface plus complexe entre le programme et la mémoire en attribuant à chaque adresse physique une nouvelle adresse virtuelle. Ce mécanisme permet de communiquer avec le programme via les adresses virtuelles et donner ainsi au programme l'illusion d'un espace d'adressage continu de la mémoire qui lui est accessible et ce même dans le cas où cette mémoire se retrouverait fragmentée à divers endroits de la mémoire physique par soucis d'optimisation. Cette solution permet également de mieux opacifier la mémoire inaccessible au programme sans qu'il puisse deviner la taille mémoire occupée par les autres programmes.

2.2.2 Système d'exploitations

Afin de faire fonctionner les composants de MPU et MMU, il est nécessaire alors d'avoir un programme de supervision chargé d'initialiser les droits d'accès à la mémoire. Ce programme est alors lancé au démarrage du système et il sera chargé d'attribuer les droits d'accès à la mémoire pour les différents programmes qui s'exécuteront par la suite. Ce programme est appelé système d'exploitation et il possède diverses fonctions qui sont séparées en deux catégories : le mode noyau et le mode utilisateur. Le noyau est le centre névralgique d'un système d'exploitation, il possède un accès non restreint à l'ensemble des ressources et c'est lui qui est responsable de l'initialisation de la MPU ou de la MMU. Le mode utilisateur est composé quant à lui d'une multitude de programmes utilitaires comme l'interface graphique ou d'autres outils nécessaires à l'utilisation d'un système. En résumé, dans le cadre d'un système complexe exécutant plusieurs programmes, le noyau est le composant logiciel privilégié qui sert d'interfaces entre les autres programmes (y compris le reste du système d'exploitation) et les ressources matérielles (dont fait partie la mémoire).

La prise en charge des fonctions de MPU et MMU est donc à la fois dépendante du matériel (la fonctionnalité doit être présente sur le matériel) et du logiciel (le logiciel doit embarquer une section de code ayant pour fonction d'initialiser la mémoire en attribuant à chaque programme ses droits d'accès).

2.3 Définition du contexte de la sécurité du flot d'exécution

Les travaux présentés dans ce document s'intéressent à la sécurité du flot d'exécution, autrement dit il visent à s'assurer que le code qui s'exécute réellement est bien celui qui a été chargé en mémoire lors du lancement du programme. Cependant certaines vulnérabilités relatives au flot d'exécution ne rentrent pas dans le scope de cette étude.

Une hypothèse prise dans les travaux présentés dans ce document est que le segment de code n'est pas modifiable en écriture durant l'exécution. En effet, la vaste majorité du matériel actuel repose sur cette garantie offerte par l'usage de mémoire exécutable persistante (ROM) ou par une gestion de la mémoire partagée (MPU, MMU). Ainsi, les attaques ciblant du matériel non protégé ainsi que les attaques physiques ciblant la RAM ou la ROM dans le but de modifier une ou plusieurs instructions ne sont pas prises en compte car leur sécurisation repose plus sur des bonnes pratiques externes de sécurité (choix du matériel, gestion des accès physiques au matériel).

Les travaux se concentrent donc uniquement sur les catégories d'attaques ciblant une architecture matérielle massivement déployée et sur des catégories d'attaquant pouvant prendre le contrôle d'un appareil sans avoir recours à une altération physique du matériel.

Dans la suite de ce document il est nécessaire de distinguer deux notions :

1. **Vulnérabilité :** Une vulnérabilité représente un défaut de conception, de développement ou d'utilisation d'un composant matériel ou logicielle pouvant permettre à un attaquant de porter atteinte à l'intégrité d'un système d'information.
2. **Exploitation :** L'exploitation d'une vulnérabilité représente quant à elle la méthode employée par l'attaquant pour porter atteinte a l'intégrité du système.

Lorsque l'on parle de sécurité du flot d'exécution, il est donc à la fois question de lutter contre la présence de vulnérabilités ainsi que de lutter contre leur exploitation. Cependant dans ce document, nous faisons le postulat qu'il existera toujours des vulnérabilités dans les systèmes et donc nous nous intéresserons à prévenir leur exploitation. C'est pourquoi les solutions de détection ou de prévention de vulnérabilités ne seront pas évaluées.

2.4 Attaque et défense du contexte d'exécution

Comme nous avons vu précédemment, le déroulement d'un programme est fortement lié au segment de code et aux différentes données qui y réfèrent. L'enjeu de la sécurité relative au flot d'exécution du programme est donc de s'assurer que ces données ne soient pas modifiées malicieusement sous peine de voir le programme être détourné de son exécution légitime.

2.4.1 Buffer overflow et injection de code.

La représentation des données en mémoire étant linéaire, les données se situent les unes à la suite des autres dans un espace délimité. Par exemple une donnée de type CHAR représentant un caractère en code ascii occupe 1 octet, une donnée de type INT32 représentant un entier signé allant de -2^{31} à $+2^{31}-1$ occupe 4 octets. Si un INT (donc 4 octets) est mémorisé dans un CHAR de 1 octet, en l'absence de mécanisme de protection l'opération va inscrire l'INT dans le CHAR ainsi que dans les 3 octets qui suivent. Cette opération est nommée dépassement de tampon ou *Buffer Overflow* en anglais.

Lors de la traduction d'un programme depuis le code source vers le code binaire (aussi appelée compilation), certains langages proposent une vérification des types de la source et de la destination d'une opération afin de s'assurer de la cohérence du résultat de l'opération. Ces langages peuvent également proposer une gestion de borne lors de l'utilisation d'objets de taille dynamique comme les tableaux. Dans notre cas nous nous intéressons principalement au langage C en raison de sa forte présence dans le développement système et plus particulièrement dans le domaine de l'embarqué. Le C est faiblement typé qui autorise les conversions implicite de type, c'est à dire qu'une même donnée peut être utilisée comme étant d'un type différent que celui défini lors de sa déclaration. Lors de la compilation d'un programme C il existe des outils (fonctionnalités du compilateur ou programmes d'analyse externe) permettant d'apporter une vérification supplémentaire et un certain contrôle des types.

De plus, dans certains langages dont le C, il existe une façon d'utiliser les données qui échappe aux contrôles de borne : les pointeurs. En règle générale le programme accède à une donnée en utilisant son identifiant. Cependant, il peut y faire référence en utilisant directement son adresse en mémoire. Cette représentation est particulièrement utile lorsque l'on veut représenter une série de données, par exemple un tableau de 35 caractères représentant un texte. Dans ce cas le pointeur est initialisé pour pointer vers la première case du tableau, puis il peut être incrémenté ou décrémenté afin de parcourir librement le tableau. Donc, lorsque l'on essaye d'écrire un texte de 50 caractères dans un tableau prévu pour 35, 15 octets supplémentaires de mémoire vont être écrasés par le texte directement via le pointeur qui continuera de s'incrémenter en parcourant la mémoire sans se soucier d'aucune notion de borne (figure 2.3).

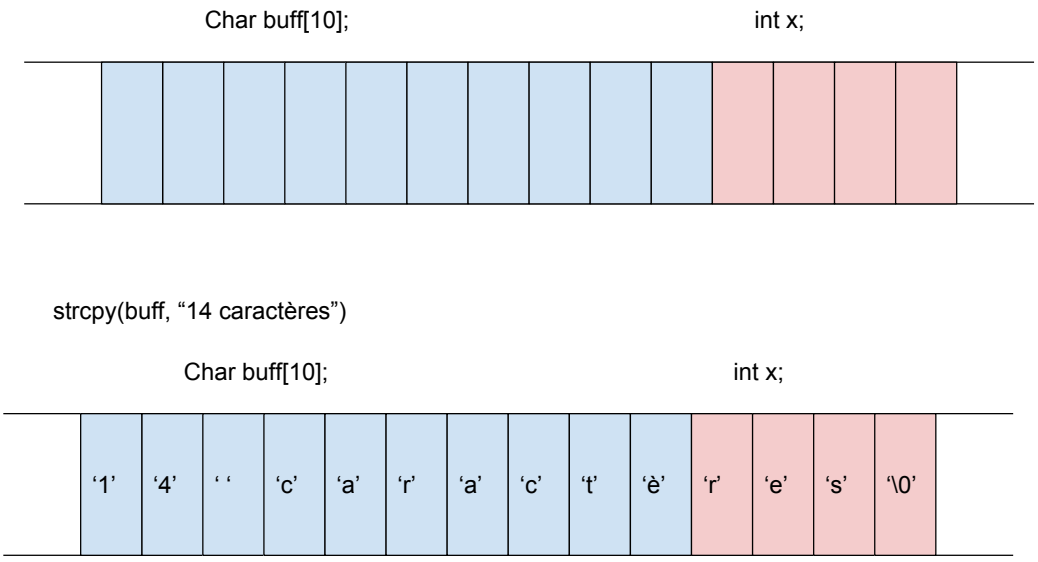

FIGURE 2.3 – Exemple de buffer overflow

Lorsque l'on parle d'action malveillante, le *buffer overflow* [77, 64, 10] consiste à exploiter ce comportement pour modifier la pile d'exécution du programme. De cette manière, il est possible de modifier le flot de contrôle de multiples manières, les plus efficaces étant de modifier la valeur de l'adresse de retour de la fonction comme illustré dans la figure 2.4. Il devient ainsi possible de faire exécuter n'importe quelle portion de la mémoire par le processeur. Si la majorité de la mémoire du système n'est pas prévue pour être exécutée, l'attaquant peut utiliser la portion de mémoire séparant la valeur initiale de son pointeur et l'adresse de retour (ou encore la mémoire se situant après l'adresse de retour) pour y stocker les instructions qu'il souhaite exécuter. Il lui suffit ainsi de faire pointer l'adresse de retour vers les instructions injectées pour détourner le programme de son comportement légitime. Une autre manière d'exploiter un *buffer overflow* est de corrompre la valeur d'un pointeur de fonction. Une fois l'adresse voulue mise dans le pointeur de fonction, l'attaque fonctionne exactement comme une corruption d'adresse de retour. Finalement d'autres attaques sont possibles depuis un *buffer overflow* comme le *printf-oriented programming* [18] ou le *String Oriented Programming* [81].

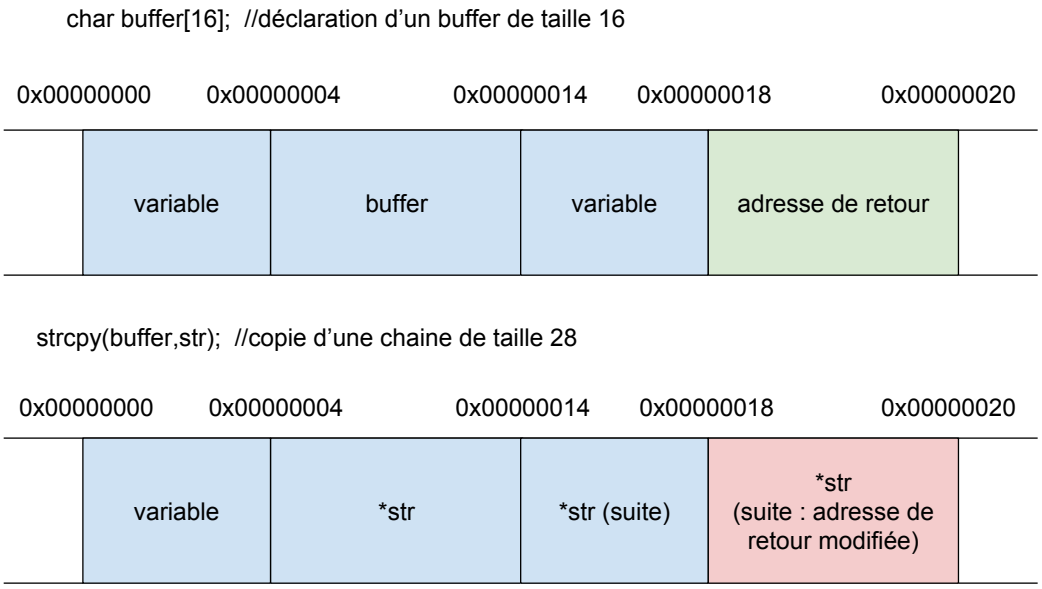

FIGURE 2.4 – Exploitation d'un buffer overflow

2.4.2 Prévention de l'exécution de donnée

En réponse aux attaques par injection de code, la solution proposée par de nombreux fabricants de microprocesseurs a été de permettre l'interdiction de l'exécution de certaines portions de la mémoire grâce à un bit dédié nommé le bit NX [2, 1]. Une fois le segment mémoire marquée comme non exécutable par le système d'exploitation, le processeur refuse d'exécuter une instruction située à une adresse marquée. L'utilisation principale de cette technique est de restreindre l'accès à la pile d'exécution. De cette manière, lorsque le compteur ordinal vient pointer vers la pile d'exécution, l'exécution est stoppée. Ce mécanisme, bien qu'efficace, possède cependant ses limites. Premièrement cette solution repose à la fois sur une implémentation matérielle et logicielle, il faut donc que le matériel et le système d'exploitation soient compatibles. Deuxièmement, si ce mécanisme empêche l'injection de code nouveau dans le programme, elle n'empêche pas de modifier le flot de contrôle ni de changer la valeur des données présentes dans la pile d'exécution.

2.4.3 Autres attaques détournant le flot de contrôle

L'exécution des données dans la pile d'exécution étant rendue impossible par les technologies telles que le bit NX, d'autres catégories d'attaques ont vu le jour visant cette fois à tirer profit du code déjà présent en mémoire afin d'exécuter des actions malicieuses, ces attaques sont appelées attaques par réutilisation de code ou *Code Reuse Attacks* en anglais.

Return-to-libc

Lorsqu'un programme s'exécute il n'utilise pas uniquement son propre code, il peut également faire appel à des librairies externes. La plus couramment utilisée sur les systèmes unix est la librairie standard C (ou libc) qui fournit des implémentations pour les tâches récurrentes (allocation mémoire, manipulation de chaînes de caractères, gestion des *threads*, etc. . .). Lors d'une attaque exploitant un *buffer overflow*, il est possible donc de modifier l'adresse de retour pour venir exécuter des fonctions provenant de ces librairies [91, 90]. La méthode la plus courante dans ce cas de figure vise à appeler la fonction "system" de la libc qui prend en paramètre un nom de commande et permet d'exécuter cette commande. En modifiant les paramètres d'appels de cette fonction, l'attaquant peut alors exécuter n'importe quelle commande présente sur le système pour effectuer des actions de lecture, écriture et de gestion des droits et le tout avec les privilèges du programme détourné. Cependant l'attaque la plus simple et la plus efficace consiste à venir lancer un interpréteur de commande (via la commande *bash* par exemple). Une fois l'interpréteur lancé l'attaquant pourra lancer toutes les commandes qu'il souhaite et donc effectuer toutes les opérations qui lui sont permises avec les droits que possède le programme vulnérable. Cette attaque peut donc permettre à un utilisateur non privilégié de s'attribuer des droits supérieurs et donc de réaliser ce que l'on appelle une élévation de privilège.

Return Oriented Programming (ROP)

Finalement le code du programme lui même peut servir à mener des attaques malicieuses, soit directement : faire pointer l'adresse de retour vers un morceau de code critique comme par exemple une fonction d'authentification, soit le plus souvent de façon beaucoup moins directe. En 2008, Buchanan & Al. [16, 87] proposent une méthode consistant à exploiter les toutes dernières instructions précédant l'instruction *return* pour constituer une chaîne de morceaux de code. Ces instructions ont pour but d'effectuer des opérations simples (lire, écrire, additionner, etc.) ainsi que des opérations plus complexes comme des boucles ou des conditions. Ces pseudo-instructions nommées gadgets peuvent prendre différentes formes et pour chaque opération souhaitée il existe une multitude de gadgets permettant de parvenir à ce résultat. Une fois les gadgets trouvés, il est possible de programmer une attaque en liant gadget entre eux (on parle alors de chaine ROP ou *ROP chain* en anglais) pour effectuer une opération plus complexe. La figure 2.5 montre un exemple de *ROP chain* permettant de modifier une valeur en mémoire. Plus la panoplie de gadgets sera complète, plus l'attaque sera efficace. Les conséquences vont de la divulgation ou corruption de la mémoire à l'élévation de privilèges et la prise de contrôle complète. La recherche de ces gadgets peut être automatisée et l'écriture de la charge utile (*payload* en anglais ou *ROP chain* dans ce cas précis), fastidieuse à écrire à la main, peut être également automatisée. L'outil transcrit alors un court programme en données binaires qui, une fois injectées dans le programme, le détourneront afin qu'il exécute le programme précédemment transcrit. Cette attaque, contrairement au ret-to-libc nécessite d'effectuer consécutivement un grand nombre de retours, elle nécessite donc de réécrire une plus grande partie de la pile d'exécution.

De plus lorsque les instructions déjà présentes ne suffisent pas à réunir les gadgets souhaités, il est possible de rechercher la présence d'instructions involontaires dans le code [20]. Dans la mémoire, les instructions sont de taille variable car celle-ci dépend

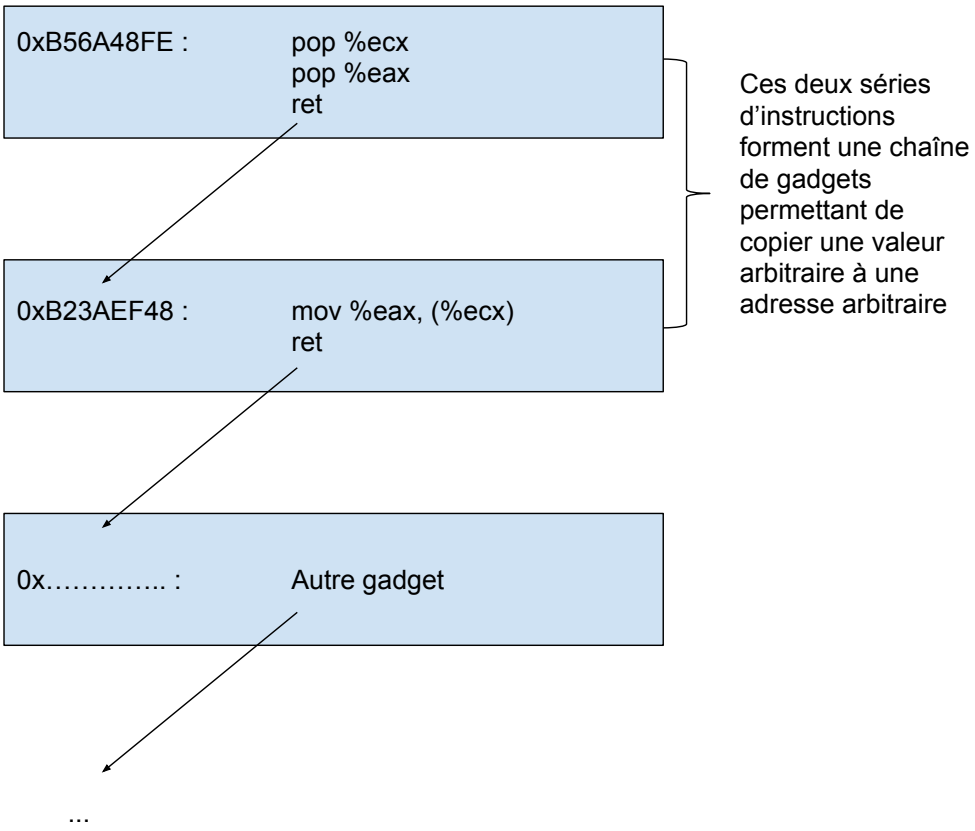

FIGURE 2.5 – Combinaison de gadgets en vue d'effectuer une action malveillante

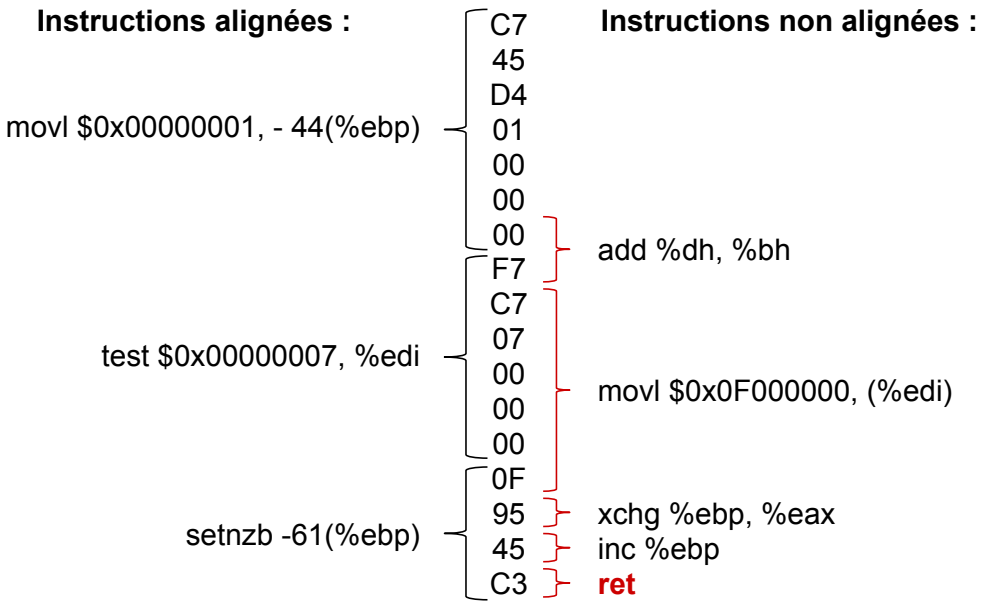

FIGURE 2.6 – Exemple de gadget non aligné

à la fois de l'instruction elle même (entre un et deux octets sur x86) mais également de ses opérandes. Ainsi une instruction interprétée comme une addition " add %eax, 10 " commençant à l'adresse 0x00000001 peut contenir une instruction tout à fait différente si l'on considère qu'elle commence à l'adresse 0x00000002, la figure 2.6 montre un autre exemple de gadget non aligné.

De cette manière il est possible d'augmenter grandement les chances de trouver des gadgets car l'on ne se contente plus uniquement des instructions *return* légitimes mais également de toutes les instructions *return* non alignées.

Finalement, des variantes de cette attaque basées sur des instructions autres que l'instruction *return* ont été proposées. Bletsch & al [12] ont proposé une implémentation basée sur l'instruction *jump* et Bosman & al. [13] sur le signal *sigReturn*.

2.4.4 ASLR et Stack Canary

L'exploitation des *buffer overflows* permet donc de profiter d'une vulnérabilité dans le code afin de détourner le comportement d'un programme et potentiellement de prendre le contrôle de la cible. Ces attaques étant basées sur le code source, leur exploitation est donc identique pour chaque cible exécutant le même code source, cette catégorie d'attaque est appelée *Break Once Run Everywhere* ou BORE soit "casser une fois, utiliser partout". Ces attaques sont donc facilement reproductibles, si l'on prend l'exemple connu des téléphones portables ou des consoles de jeu, il est donc possible de distribuer à grande échelle une solution permettant d'obtenir le contrôle total sur ces appareils (et donc d'exécuter du logiciel non autorisé, ou des jeux piratés dans le cas des consoles de jeu).

Afin de limiter l'impact des *buffers overflows*, des défenses probabilistes ont été mises au point. Ces défenses ne sont pas des défenses absolues mais ont pour objectif de complexifier la tâche de l'attaquant en rendant chaque défense unique et donc en obligeant une attaque personnalisée pour chaque appareil.

Stack Canary

La première est appelée *Stack canary* [28, 101] en référence aux canaris utilisés dans les mines de charbon pour détecter les fuites de poches de gaz. Ces derniers étant plus vulnérable mourraient rapidement laissant ainsi aux mineurs le temps de sortir. Le principe de ce mécanisme logiciel est l'insertion d'une valeur de contrôle placée juste avant l'adresse de retour dans la pile d'exécution. Lors d'un retour de fonction la valeur de contrôle est vérifiée et le programme s'arrête en cas d'altération du canari plutôt que d'effectuer le retour (voir figure 2.7). L'attaque, pour réussir, nécessite de trouver la valeur du canari (l'attaquant n'a obtenu que la possibilité d'écrire dans la mémoire, mais pas d'y lire). Pour complexifier encore la tâche de l'attaquant il existe 3 types de canaris.

Les canaris aléatoires [50], ceux ci prennent une valeur aléatoire attribuée à l'exécution et constituent une défense probabiliste en forçant l'attaquant à essayer de les deviner.

Les canaris XOR consistent à effectuer une succession d'opérations XOR entre une valeur aléatoire et les données de contrôle (valeur de la graine aléatoire et valeur de l'adresse de retour). Grâce à cette opération il est possible de savoir si le canari ou les données de contrôle ont été modifiés. La tâche de l'attaquant sera donc de deviner la graine aléatoire, mais également l'ensemble des opérations effectuées sur cette graine afin de reproduire ces opérations sur la graine et la nouvelle adresse de retour.

Le canari terminal n'est quant à lui pas une défense probabiliste et est donc vulnérable au BORE. Il a pour valeur un caractère terminal de chaîne : '\0'. Ainsi si le *buffer overflow* est construit à partir de copie de chaînes de caractères et que l'attaquant souhaite recopier le canari à l'aide d'un fonction comme strcpy() alors cette fonction se terminera automatiquement lorsqu'elle rencontrera le caractère terminal et donc sans écraser l'adresse de retour. En revanche si ce canari fonctionne contre les manipulations de chaînes de caractères, il est inefficace contre les autres *buffer overflow* car sa valeur fixe est connue.

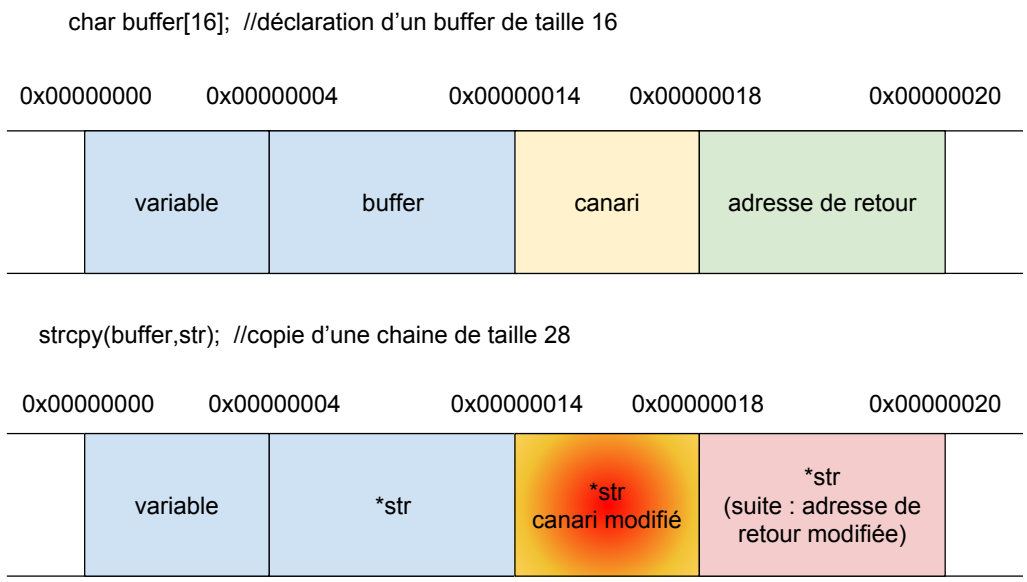

FIGURE 2.7 – Mécanisme de protection de type canari

A.S.L.R.

La deuxième solution déployée est l'ASLR [66] pour *Address Space Layout Randomization* ou distribution aléatoire de l'espace d'adressage. Cette solution, uniquement logicielle, consiste à organiser de manière aléatoire l'espace mémoire dans lequel sont stockés les librairies partagées, la pile d'exécution et le segment de code. Ainsi lors de l'exploitation d'un *buffer overflow*, l'attaquant doit deviner l'emplacement [71] du segment de code et des librairies vers lesquels il souhaite détourner le flot de contrôle. A noter que certaines librairies ne sont pas concernées par la distribution aléatoire des implémentations courantes d'ASLR, ce qui permet à l'attaquant de monter quand même une attaque [89].

Combinaison

Ces deux défenses peuvent être combinés afin de complexifier la tâche de l'attaquant en rendant l'exploitation du *buffer-oveflow* non triviale, non automatisable et surtout détectable. L'attaquant qui souhaiterait contourner un canari et l'ASLR devra exécuter le programme un grand nombre de fois pour parvenir à deviner la graine aléatoire, l'ensemble des opérations effectuées par le XOR-canari ainsi que l'emplacement de la libc afin de prendre le contrôle du système. Ces défenses ne sont donc pas absolues mais constituent une contre-mesure probabiliste rendant l'exploitation de la vulnérabilité hautement plus difficile pour l'attaquant pour un surcoût d'exécution faible.

2.4.5 Control-flow integrity

Return adresse protection

En complément aux deux contre mesures décrites précédemment, d'autres solutions de mitigations ont été proposées pour répondre au problème de détournement de flot de contrôle.

Premièrement, les défenses de type *shadow stack* [48, 30] qui consistent à créer une copie de l'adresse de retour et de la cacher en mémoire, au moment du retour les deux adresses sont comparées à la recherche de modification. Cette défense fonctionne sur le même principe que le canari, cependant contrairement au canari le duplicata peut être stocké n'importe où, ainsi la copie peut être stockée dans un registre du processeur ou dans des zones protégées de la mémoire (ARM *trustzone* [40]/Intel SGX [26]). Une autre alternative est celle proposée par Grsecurity [48] qui se base sur un chiffrement asymétrique de l'adresse de retour sauvegardée. La sauvegarde est chiffrée à la compilation avec une clé privée, l'adresse d'origine elle n'est pas modifiée. Lors du retour, l'adresse sauvegardée est déchiffrée avec la clé publique et comparée à l'adresse de retour effective. Ainsi même si l'emplacement de la sauvegarde est trouvé, l'attaquant doit pouvoir chiffrer la nouvelle adresse de retour avec une clé privée qui n'est pas présente sur le système. Ces solutions protègent la sauvegarde de l'adresse de retour contre la corruption et donc empêchent d'exploiter un *buffer overflow* par une adresse de retour. Les attaques visant à modifier un pointeur de fonction restent par contre fonctionnelles même en présence d'une *shadow stack*.

Code-pointer Integrity

Afin de résoudre le problème posé par la corruption des pointeurs de fonctions (aussi appelé problème des branchements indirects), des solutions [59, 109, 99] ont été proposées pour protéger également la valeur de ces pointeurs. La problématique est que contrairement à l'adresse de retour qui est fixée, l'adresse d'un pointeur de fonction à pour vocation d'évoluer au fil de l'exécution. Ainsi il n'est souvent pas possible de prédire précisément [84] la valeur du pointeur lors de son appel (c'est par définition l'intérêt même du pointeur de fonction). Les solutions de CPI utilisent l'analyse statique et l'interprétation abstraite pour restreindre les cibles potentielles à un sous-ensemble et ainsi réduire la surface d'attaque.

Contrôle de transitions

Une autre approche consiste à vérifier de manière plus générale les transitions du flot d'exécution. Un profil de l'application, consistant généralement en un graphe représentant de flot de contrôle (*Control Flow Graph* en anglais ou CFG) du programme de façon plus ou moins précise, est extrait lors de la compilation afin de lister toutes les transitions valides entre les éléments du programme. Ces vérifications peuvent intervenir à différents niveaux de granularité (transitions entre module, fonctions ou blocs de base) et peuvent être plus ou moins complètes : vérification avant/après l'entrée dans une fonction et vérifications avant/après son retour. Cette méthode a été introduite par Wagner & al. [102] puis par reprise par d'autres [108, 65, 7] avant d'être présentée dans sa version la plus connue qui a été formalisée par Abadi et al.[3, 4]. Depuis un grand nombre de solutions [79, 98, 110, 67, 75, 76, 74, 70, 27, 43, 56, 55]

ont été proposées proposant une granularité plus ou moins grande, avec des contrôles plus ou moins complets et un modèle de référence plus ou moins précis le tout afin de trouver un compromis entre sécurité et surcoût d'exécution.

Le CFI parfait

Dans l'absolu un CFI à fine granularité, surveillant chaque transition du flot de contrôle, utilisant la cryptographie pour sauvegarder les valeurs de références et se basant sur un modèle précis du comportement du programme, devrait offrir une garantie de la bonne exécution du programme. Cependant même si cela était réalisable, un tel CFI aurait un coût non acceptable pour le programme à protéger [27].

2.4.6 Limites du CFI

Les limites du CFI dépendent majoritairement de son implémentation, cependant on peut distinguer trois grandes familles de problèmes impactant le CFI [78].

Premièrement les problèmes liés aux attaques contre le CFI en lui même. Les mécanismes de CFI proposés évoluent dans le même environnement que le programme vulnérable et doivent donc être dissimulés, que ce soit pour cacher une sauvegarde d'adresse de retour, une liste de pointeurs ou un table de transitions légitimes. Ces données sont donc vulnérables aux attaques par fuite de mémoire[25] qui consistent à lire ou deviner tout ou partie de la mémoire. Seules les défenses à base de chiffrement asymétrique ne sont pas concernés par la fuite de mémoire[48], cependant leur mise en place est coûteuse et ne peut pas entrer dans le cadre d'un CFI parfait.

Deuxièmement les problèmes liés à la couverture du CFI, il est difficile de prédire avec exactitude les cibles d'appels indirects et donc fournir une couverture totale du code à protéger. Ces imperfections ont donné lieu à de nombreux exemples d'attaques [36, 31, 17] se basant principalement sur l'effet de "fonction de distribution". Ces fonctions, appelées à de nombreux endroits du code, permettent à l'attaquant de passer de l'endroit où se situe la vulnérabilité jusqu'à une fonction critique (figure 2.8).

Finalement, les attaques de type -itdata-only [52], les politiques de CFI s'intéressent uniquement à détecter des transitions illégitimes, c'est à dire une corruption d'appel de fonction ou d'adresse de retour. Or lorsque l'on exploite un *buffer overflow* il est possible de modifier des données qui ne sont pas des données de contrôle. Par exemple dans une fonction appelant la commande "execve", il est possible de modifier la chaîne qui est passée en paramètre à la commande de la remplacer par "/bin/bash". Il est également possible de modifier la valeur du flag "is_admin" pour la passer de 0 à 1 ou plus généralement de changer la valeur d'une variable pour passer un branchement conditionnel. Ces attaques sont beaucoup plus dépendantes du contexte du programme mais elles sont également extrêmement difficiles à détecter car elles ne modifient pas les données de contrôle, pour autant elles peuvent mener à une modification effective du flot de contrôle. Des récentes études commencent à proposer des solutions à ce problème en essayant d'identifier automatiquement les données critiques afin de les protéger [92].

Cependant il faut être conscient que les vulnérabilités exploitables de type *buffer overflow* ne sont pas présentes à tous les endroits du programme. De plus, la portion de mémoire modifiable lors d'un *buffer overflow* est limitée à la mémoire située en amont

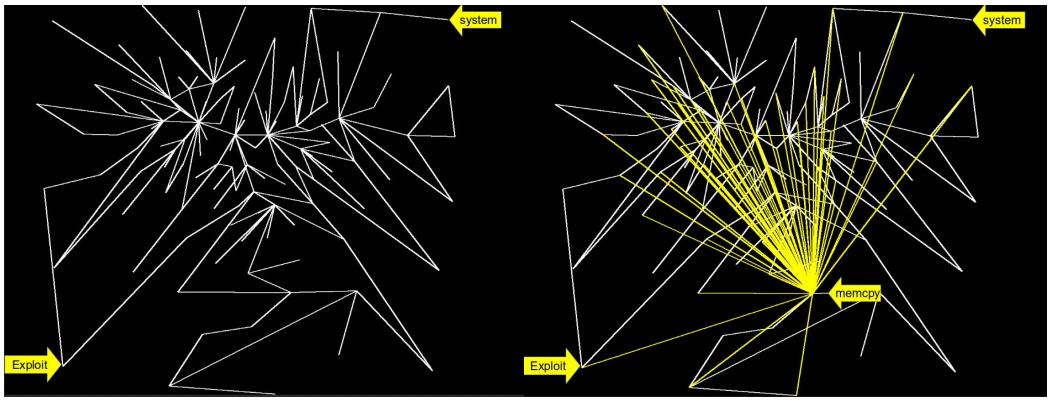

FIGURE 2.8 – Control Flow Bending : Utilisation de la fonction memcpy comme fonction de distribution pour relier la fonction vulnérable aux sections critiques du programme

du *buffer overflow*. De ce fait le besoin de modifier le flot d'exécution du programme est primordial pour l'attaquant afin de pouvoir faire le lien entre la vulnérabilité (souvent située dans une portion peu testée du code car non critique) et les sections critiques du programme qui sont généralement testées et ne présentent pas de failles de ce type. C'est pourquoi, même si les attaques de type *data-only* ne sont pas directement couverte par les défenses de types CFI, elles sont quand même fortement mitigé en isolant les différentes parties du programme entres elles et en restreignant la surface d'attaque à l'environnement proche de la vulnérabilité.

2.5 Sécurité et développement de systèmes embarqués

Maintenant que nous avons dressé l'état de l'art en matière de sécurité du flot d'exécution, il est nécessaire de s'intéresser à l'état des lieux sur les systèmes embarqués.

2.5.1 DEP et ASLR

Les processeurs ARM proposent l'utilisation de la prévention de l'exécution des données depuis les version ARMv6 (2009), les processeurs Intels depuis les versions Pentium 4-prescott et Xeon-Nocona (2004) et sur les processeurs AMD depuis Opteron et Athlon (2003)

Cependant, cette contre-mesure matérielle doit s'accompagner d'une implémentation logicielle afin d'être effective. Afin de permettre une implémentation de la DEP, les systèmes embarqués doivent proposer une implémentation des mécanismes de protection mémoire (MPU ou MMU). Ainsi on trouve un support de la DEP et de l'ASLR dans les systèmes QNX, µOS, wind river linux et VxWorks. Les systèmes plus légers tels que Contiki OS, Tiny OS, FreeRTOS ou icOS sont principalement destinés à du matériel n'offrant pas de mécanismes de protection mémoire et donc n'embarquent pas de contre-mesures logicielles au détournement de flot contrôle.

2.5.2 Canari

Le déploiement d'un protecteur de pile de type canari est une solution purement logicielle et se met en place automatiquement lors de la compilation, elle est disponible sous gcc depuis la version 3.0 et dans clang depuis la version 6.0. Cette option de compilation peut donc tout à fait être déployée sur des systèmes embarqués d'autant plus que son coût est minimal.

2.5.3 Control Flow Integrity

La mise en place de solutions d'intégrité de flot de contrôle sur les systèmes embarqués est à décomposer en deux catégories. La première catégorie est le déploiement de solutions non spécifiques à l'embarqué comme le patch PaX proposé par grsecurity [48] pour linux ou les solutions proposées par LLVM [99]. Ces solutions posent un sérieux problème de performance sur des systèmes très contraints mais également un problème de sécurité car celles-ci s'appuient sur des fonctionnalités de sécurité telles que l'ASLR, l'isolation mémoire et la prévention de l'exécution des données. Or comme vu dans la section 2.5.1 ces fonctionnalités ne sont pas toujours implémentées par les systèmes d'exploitation dédiés à l'embarqué qui privilégient plutôt des fonctionnalités de performance et de sûreté d'exécution.

La deuxième catégorie regroupe les solutions conçues spécifiquement pour l'embarqué [39, 83]. Cependant ces solutions reposent souvent sur une implémentation matérielle en plus du logiciel. Si elles peuvent être pertinentes pour des systèmes ultra critiques, leur déploiement dans l'industrie pour des appareils massivement déployés reste toutefois peu probable car cela nécessiterait une modification de la chaîne de production et donc une hausse du coût de déploiement. Il semble donc plus réalisable de s'orienter sur une solution s'intégrant directement de le processus de développement déjà en place en la rendant compatible avec les matériels et systèmes présents sur le marché. C'est pourquoi les solutions intervenant au moment de la compilation semblent particulièrement appropriées.

2.6 Stratégies d'implémentation des mécanismes de défense

Afin de palier aux problèmes de sécurité énoncés précédemment, plusieurs stratégies sont possibles, certaines comme le *fuzzing* et l'analyse de code ont pour but de détecter les vulnérabilités, ce sont des stratégies proactives, d'autres comme l'instrumentation de code et la supervision sont des stratégies réactives ayant pour but de détecter non pas la vulnérabilité mais l'attaque qui cherchera à l'exploiter. Finalement les défenses ayant pour but de renforcer le système afin de rendre toutes ou certaines attaques plus compliquées à mettre en place, celles-ci s'appuient généralement sur de l'instrumentation de code ou sur une fonctionnalité du matériel.

Toute ces stratégies s'inscrivent dans un cycle de développement qui peut être formalisé. Un exemple d'une telle formalisation est le Microsoft Security Development Lifecycle (MSDL) présenté dans la figure 2.9 qui découpe la mise en place d'un processus de sécurité en 7 phases.

Les phases qui nous intéressent particulièrement sont les phases 4 à 6 : Implémentation, vérification et déploiement.

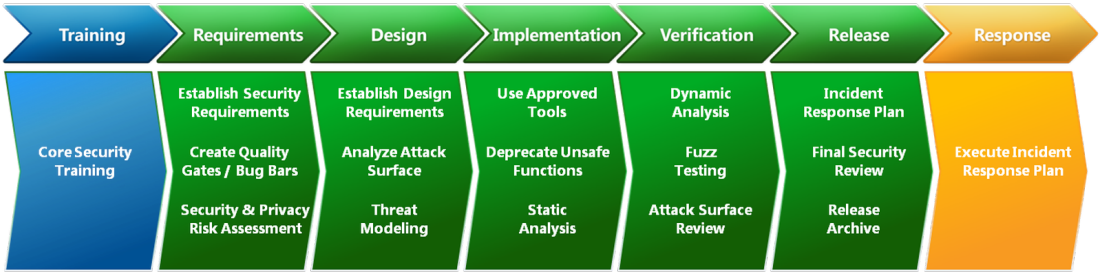

FIGURE 2.9 – Microsoft Security Development Lifecycle (MSDL)

2.6.1 Implémentation

Le développement d'une stratégie de sécurité lors de l'implémentation peut s'effectuer à deux niveaux indépendants et complémentaires : dans le code source et dans le code binaire.

Le premier cas concerne le développement d'applications en utilisant des bonnes pratiques de sécurité. Ainsi une liste non exhaustive des bonnes pratiques de développement pour la sécurité comprend par exemple :

- L'application du principe de moindre privilège avec une politique de contrôle d'accès
- L'utilisation d'un langage adapté aux besoins.
- L'utilisation d'un typage strict, même si le langage ne l'est pas.
- L'utilisation de fonctions standards sécurisées (on préférera par exemple utiliser strncpy() plutot que strcpy() afin de s'assurer du nombre maximal de caractères à copier).
- L'utilisation de fonctions d'échappement des entrées permettant de s'assurer qu'une variable ne contient pas de code exécutable (pour éviter les injections SQL par exemple)

Mais l'application de ces bonnes pratiques est entièrement soumise au bon vouloir et à la compétence du développeur. Dans l'absolu, celle-ci devrait permettre d'éviter les vulnérabilités, mais en pratique un programme complexe peut représenter plus d'un million de lignes de code écrites par des centaines de développeurs partout dans le monde sur une période pouvant s'étaler sur plusieurs dizaines d'années. Dans ce contexte, il est absurde de penser qu'aucune erreur ne peut être commise, et un travail consciencieux des développeurs peut au mieux permettre de limiter fortement le nombre de vulnérabilités sans pour autant pouvoir les éliminer de manière certaine.

Afin de compléter le travail fourni par les développeurs, mais également afin de contrôler le travail effectué par un tiers (sous traitant, système d'exploitation, brique logicielle externe, librairies, applications utilisateur, etc. . .) il est possible d'effectuer des vérifications et des améliorations à posteriori du développement.

Le premier exemple est l'utilisation d'un compilateur efficace pouvant fournir des options de sécurité lors de la compilation (*stack canary*, *bound checking* [6, 72, 82], *control flow integrity* [99]). L'instrumentation de code est l'outil principal du déploiement de solutions de sécurité du flot de contrôle, utilisée soit pour intégrer des mécanismes de détection d'intrusion, soit pour renforcer le code contre les attaques. L'instrumentation est à la base de solutions telles que le canari, le CFI et le CPI. Cette instrumentation peut se faire directement dans le code source (on parle alors de patchs de sécurité) ou lors de la compilation. Le *framework* LLVM permet également d'instrumenter un code

déjà compilé en transformant un binaire en langage intermédiaire sur lequel il va être possible d'ajouter des instructions avant de le recompiler.

Ensuite il est possible de procéder à des analyses sur le binaire avec des outils tels que Valgrind [73] ou Gprof [37]. En effet les vulnérabilités ont souvent une signature commune et reposent souvent sur des mécanismes similaires. Par exemple un *buffer overflow* repose souvent sur une écriture sans contrôle de la taille avec par exemple l'appel de la fonction strcpy à la place de la fonction strncpy qui elle effectue un contrôle sur la taille de la donnée à copier. L'analyse de code peut donc permettre de détecter des vulnérabilités avérées mais également des comportements à risques pouvant donner lieu à des vulnérabilités. Une fois détectés ces points vulnérables du programme peuvent éventuellement être patchés afin de renforcer le programme [21] en modifiant le code sensible par un code plus sécurisé.

L'analyse de code peut également être utilisée pour établir un profil de l'exécution, c'est le cas notamment dans les solutions de CFI ou elle est utilisée pour extraire un graphe de flot de contrôle qui servira ensuite de témoin pour définir si l'exécution est légitime ou dans le CPI pour prédire les cibles possibles d'un branchement indirect. Finalement elle sert aussi à détecter les gadgets dans le cas du ROP que ce soit pour réduire la surface d'attaque ou pour évaluer le potentiel de ces attaques.

2.6.2 Vérification

La mise en place d'une attaque repose souvent sur l'exploitation d'un bug dans le code, une fonctionnalité qui dans un cas limite n'agit plus de manière voulue peut permettre à l'attaquant de tirer ce comportement arbitraire à son avantage. L'exemple le plus courant est le *buffer-overflow*, mais cependant ces vulnérabilités peuvent prendre bien d'autre forme comme c'est le cas par exemple pour les injections SQL ou les injections de commande qui consistent à introduire du code dans une chaîne de caractère qui sera ajoutée à du code (requête SQL, commande shell). Un exemple encore plus simple consiste à envoyer des données non valides pour tenter de provoquer une erreur non gérée et ainsi causer un crash du système.

Ces bugs, souvent peu impactant pour le déroulement du système dans des conditions normales d'utilisation, peuvent se retrouver critiques s'ils sont exploités par un utilisateur malveillant comme dans le cas des attaques par *buffer-overflow*. Afin de les détecter, il existe plusieurs solutions, la première est de tester le code afin de s'assurer que tous les cas de figure possibles ont étés envisagés, ces tests peuvent concerner le fonctionnement global du système (tests fonctionnels) ou s'orienter vers le test d'une partie précise du code source (tests unitaires). Cependant ces tests sont réalisés le plus souvent manuellement et ont donc peu de chance de couvrir un cas de figure qui n'a pas été envisagé par les développeurs. Afin de résoudre ce problème des logiciels proposent de tester une à une toutes les possibilités afin de trouver celles qui n'auraient pas été envisagées par le développeur. Cette méthode appelée *fuzzing* [49, 46, 45, 97] peut elle aussi être fonctionnelle : les entrées du logiciel sont testées avec toutes les valeurs possibles. Elle peut également être localisée lorsqu'elle est appliquée par exemple les paramètres passés à une fonction. Finalement le *fuzzing* peut être guidé, en effet lors du *fuzzing* d'un programme complexe ou possédant des entrées non triviales (une chaîne de caractère de longueur indéterminée par exemple) il est souvent impossible de parcourir toutes les possibilités et il devient nécessaire de guider le *fuzzer* dans sa tache afin de tester des entrées plus cohérentes. Un *fuzzing* guidé pourra par

exemple avoir recours à un solveur de contrainte [54] ou l'interprétation abstraite [94] afin de trouver les paramètres nécessaires pour couvrir l'intégralité des possibilités du programme, ou se baser sur une base de données de valeurs sensibles en particulier pour détecter des possibles injections de commande dont la syntaxe bien particulière est peu probable d'apparaître spontanément dans une génération purement aléatoire.

2.6.3 Déploiement

Enfin lorsque le programme est déployé il ne reste plus qu'à s'assurer que tout se déroule correctement. Cette action appelée *monitoring* consiste à effectuer de la détection d'intrusion (et potentiellement agir en conséquence) en observant l'exécution du programme. Ces solutions peuvent êtres internes [22], directement depuis le système d'exploitation [9] ou en ajoutant une couche supplémentaire comme lors de l'utilisation d'un débogueur ou d'un hyperviseur [103]. Ces solutions peuvent également être externes grâce à la production de traces d'exécution soit en se branchant directement au matériel, soit en s'assurant que le matériel produise une trace (rapports d'erreurs, logs, trace d'exécution).

2.6.4 Autres phases du MSDL

En amont de ces 3 phases on retrouve la sécurité pré-développement : évaluation des besoins de sécurité, évaluation des risques, intégration de la sécurité dans le design du programme. Ces trois étapes sont spécifiques au logiciel/matériel utilisé et doivent impérativement faire partie du cycle de développement de chaque logiciel. Il est inutile de faire de la sécurité pour un appareil qui possède des failles conceptuelles importantes. Un exemple concret d'échec de la sécurité pré-développement est l'utilisation de logins/mot de passe par défaut dans des appareils embarqués massivement déployés ayant conduit à la création de *botnets* ou encore l'ajout d'une porte dérobée (*backdoor*) dans un logiciel, qui seront utilisables aussi bien par le fournisseur de logiciel que par un attaquant bien informé. Dans ces deux cas l'attaquant ne fait en aucun cas une utilisation détournée du programme, se contentant d'exploiter la paresse ou la prétention du développeur pour prendre le contrôle de sa cible.

Finalement la dernière étape est la réponse aux incidents. Cette étape dépendra également des spécificités métier du logiciel mais également de spécificités juridiques. Si celle-ci devrait s'intégrer entièrement au reste de cycle de développement, elle reste toutefois souvent négligée voire, dans beaucoup de cas dans l'informatique embarquée, complètement impossible lorsque les appareils ne possèdent aucun mécanisme de mise à jour.

2.7 Conclusion

Après avoir passé en revue l'historique général de l'évolution des techniques d'attaque et de défense du control flow plusieurs leçons peuvent être tirées.

Premièrement, à propos de la défense elle même, les différentes solutions proposées ont montré qu'il est nécessaire de faire un compromis entre couverture de surveillance et performance. De plus il a également été mis en évidence qu'il était non trivial et même parfois impossible de prédire le comportement attendu d'un programme avec

précision en utilisant l'analyse de code. Cette imprécision peut donner à l'attaquant une marge de manœuvre suffisamment grande pour mener à bien des attaques complexes à partir d'opérations simples.

Deuxièmement, à propos de l'implémentation, les différentes attaques qui ont été menées sur les solutions de défense actuelles ont montré que même en dissimulant les données de contrôle dans la mémoire, il est possible de deviner leur emplacement et d'attaquer la solution de défense.

Finalement, en ce qui concerne les appareils embarqués, les solutions de sécurité proposées reposent sur des approches matérielles qui offrent une employabilité industrielle très restreinte, car elles modifient le processus de développement.

Chapitre 3

Problématique

Ce chapitre détaille dans un premier temps le développement d'une solution de sécurité adaptée à l'informatique embarquée et ses spécificités. Cette solution, nommée EE-CFI, consiste en une externalisation du moniteur de surveillance dans le but de permettre l'adaptation d'une solution classique de CFI aux systèmes embarqués. De plus, cette externalisation permet également de limiter fortement les attaques ciblant le système de CFI.

Dans un second temps, ce chapitre revient plus en détail sur les opportunités offertes par les systèmes embarqués pour résoudre les problèmes de précision du CFI tel qu'il est proposé actuellement. Une solution utilisant l'apprentissage automatique afin de déduire le comportement du programme en observant son exécution est proposée. Cette solution aura pour but de répondre aux problématiques de branchements indirects et de réduire l'imprécision du CFI de façon plus générale.

3.1 Éléments d'architecture du CFI

Comme il a été vu dans le chapitre précédent, les solutions d'intégrité du flot de contrôle (CFI) peuvent être variées tant en terme de design qu'en terme d'implémentation. Cependant ces solutions reposent sur un ensemble de composants pouvant être identifiés et formalisés.

3.1.1 Avant l'exécution

Afin de préparer la surveillance du flot de contrôle, il est souvent nécessaire d'analyser le code source et/ou de le modifier. La figure 3.1 montre comment les différents éléments interagissent entre eux afin d'extraire les données nécessaires à la surveillance et/ou de renforcer le code.

Cible : c'est le logiciel que l'on veut protéger. Cette cible peut représenter l'ensemble du code exécuté ou une sous partie.

Code source : le code source est la base du programme, en général le code source n'est pas modifié mais son analyse peut servir de base à la compréhension du comportement attendu du programme.

Chaîne de compilation : Celle-ci permet de traduire un code source en code machine. Elle est souvent utilisée comme moyen d'introduire tout ou partie de la solution de sécurité dans le programme via la phase d'instrumentation qui

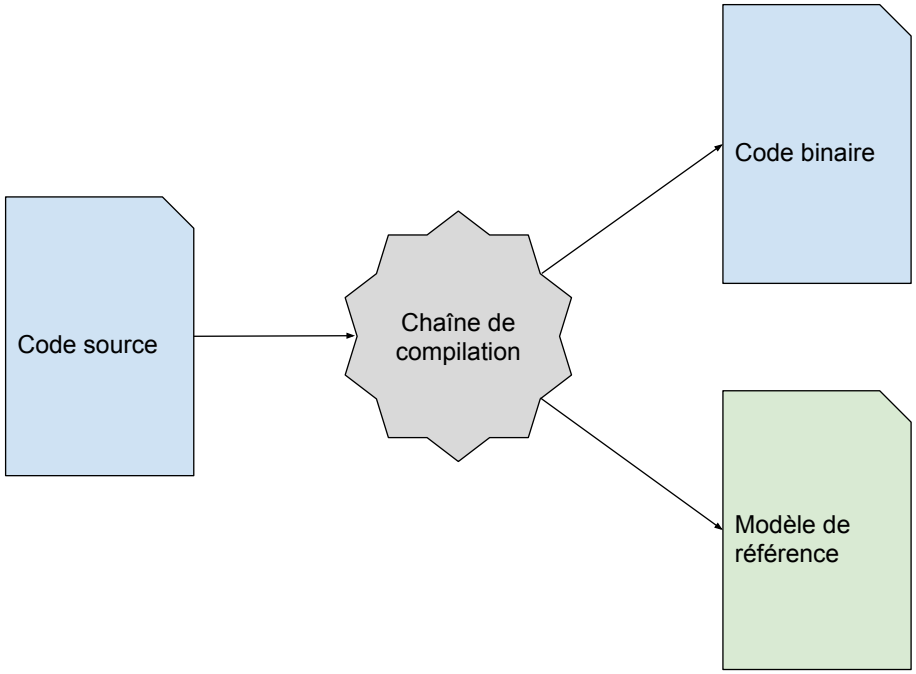

FIGURE 3.1 – Extraction du modèle avant l'exécution

consiste à ajouter de façon automatique des instructions dans des endroits choisis du code. Par exemple, la phase d'instrumentation permet d'ajouter des instructions de vérification d'un canari avant chaque instruction return. La chaîne de compilation permet également d'effectuer des analyses plus poussées sur le comportement du programme grâce à une modélisation interne et des outils d'analyse pré-existants.

Code binaire : C'est la représentation finale du programme, ce code est celui qui sera chargé lors de l'exécution et lu par le processeur. Il peut être la traduction exact du code source en code machine ou bien être une version transformée incluant des mécanismes de sécurité ou de surveillance.

Modèle de référence : C'est le modèle qui servira de point de comparaison pour la vérification. Ce modèle peut être très simple (copie des adresses de retour, cibles de branchement indirects) ou plus évolué (cartographie des appels système, graphe de flot de contrôle).

3.1.2 Pendant l'exécution

Lors de l'exécution, les différents mécanismes de sécurité entre en activité et peuvent interagir avec le programme à plusieurs niveaux :

Supervision matérielle : C'est le matériel qui vérifie l'exécution, celui-ci peut être dédié à cette tache ou il peut s'agir d'une fonctionnalité présente sur un matériel multi-usages comme c'est le cas par exemple pour la prévention à l'exécution des données.

Supervision logicielle : L'exécution est surveillée par un autre programme. Celui-ci peut être plus privilégié (par exemple le système d'exploitation ou un logiciel

de supervision en cas de virtualisation) et faire de la supervision directe, il peut également s'agir d'un autre programme au même niveau de privilège qui supervise à l'aide de traces d'exécution.

Surveillance interne : Le programme est modifié pour s'auto-surveiller. C'est le cas notamment des solutions de type canari qui vérifient la valeur du canari avant chaque retour.

Le schémas 3.2 représente le positionnement des trois niveaux de supervision par rapport aux différentes couches de l'exécution d'un programme.

Environnement d'exécution : A la fois logiciel et matériel, l'environnement d'exécution représente tout le support nécessaire à l'exécution du programme. Les solutions de sécurité peuvent agir sur celui-ci afin de surveiller le programme de l'extérieur que ce soit en passant par le matériel, le système d'exploitation ou un outil de supervision (superviseur, virtualisation).

Contexte d'exécution : Le contexte d'exécution est la réalité du programme au moment de son exécution. Il est composé de toutes les données statiques et dynamiques nécessaires à son exécution.

Politique de protection : C'est ce qui va être surveillé. Cette protection peut être ponctuelle (adresse de retour, branchements indirects) ou plus générale (surveillance des appels systèmes, surveillance du flot de contrôle). La politique de protection est décomposée en deux étapes : la surveillance et la réaction.

Moniteur de surveillance : C'est le logiciel (ou la fonction du logiciel) responsable de la surveillance effective. Celui-ci peut être interne (i.e. intégré au code binaire) ou externe (i.e. intégré à l'environnement d'exécution).

Moteur de réaction : C'est le logiciel capable de prendre une décision lorsqu'une anomalie est détectée et d'agir en conséquence.

Lien moniteur : Lorsque le moniteur et le moteur de réaction sont distincts, ceux-ci communiquent via un lien qui peut être interne (appel système, librairie externe) ou externe (envoi d'un rapport vers l'extérieur).

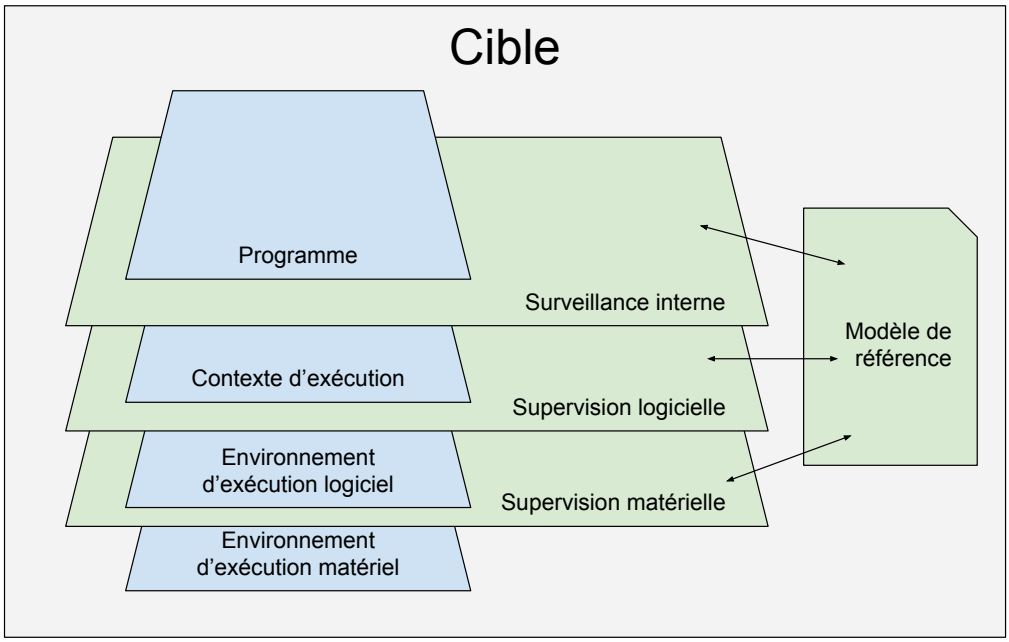

FIGURE 3.2 – Surveillance du programme pendant l'exécution

3.2 EE-CFI : CFI Externalisé pour l'Embarqué

L'objectif de EE-CFI est de proposer une solution cohérente de sécurité pour les systèmes embarqués en externalisant le processus CFI. Cette externalisation présente deux avantages majeurs. Premièrement l'externalisation permet l'isolation physique du système de surveillance et du modèle de référence. Cette isolation offre une garantie d'intégrité et de confidentialité sur les données critiques de la politique de protection. Deuxièmement, cette séparation permet de palier aux problèmes de performance posés par le portage du CFI sur les système embarqué en limitant la tâche effectuée par la cible au strict minimum.

3.2.1 Différentiation cible/moniteur et surveillance interne

L'externalisation de la solution de CFI est réalisée en séparant physiquement le moniteur de surveillance et la cible. Cependant, le contexte des systèmes embarqués et leur faible sur-couche logicielle rendent impossible l'utilisation de solution de supervision logicielle. De plus, la variété de matériel cible empêche également de considérer une solution de supervision par le matériel. La solution doit donc être une solution de surveillance interne. Afin de réduire au maximum la tâche de la cible, le code est instrumenté afin de produire une trace de son exécution et de l'envoyer vers le moniteur de surveillance. Le moniteur se situe à l'extérieur du système et se charge de vérifier si la trace correspond au modèle de référence. Cette externalisation a bien sûr de nombreux impacts sur la cible et sur sa surveillance. Cette section détaille les impacts directs et indirects sur la sécurité et sur les performances de cette externalisation.

Impact sur la sécurité

La différentiation entre la cible et moniteur offre plusieurs avantages en terme de sécurité, en premier lieu elle offre une isolation physique de la mémoire qui contient les données de sécurité. Cette isolation permet d'éviter les attaques basées sur la connaissance du modèle de référence ainsi que celles basées sur la corruption de la pile d'exécution du dispositif de défense.

Ensuite, cette alternative permet une meilleure gestion en cas de mise hors service de la cible. Dans le cas où le dispositif cesserait de fonctionner, une solution de défense interne pourrait également être mise hors service. A l'inverse il est également possible que ce soit la solution de défense elle même qui soit mise hors service alors que l'appareil cible fonctionne encore. L'externalisation permet d'identifier plus précisément le problème car l'analyse de la trace produite avant l'arrêt du dispositif peut permettre de savoir pourquoi le dispositif s'est arrêté (panne matérielle, logicielle ou action malveillante).

Finalement, cette solution permet de conserver un historique d'exécution afin de réaliser une analyse de crash pour identifier les vecteurs d'attaque ou d'instabilité du système. Lors d'une attaque, le moniteur possède une copie du flot de contrôle. Il est ainsi possible de savoir à quel endroit du code l'attaque a lieu et potentiellement quelles ont été les actions effectuées par l'attaquant.

Cette solution impose néanmoins des restrictions fortes sur la stratégie de défense. L'externalisation implique un délai entre l'exécution et la vérification qui est dû au coût temporel de la communication externe et des politiques de cache au niveau de la cible. Ce délais empêche notamment les solutions qui visent à une prévention de l'exécution malveillante (via un kill du processus corrompu par exemple). Mes travaux s'inscrivant dans une démarche de détection d'intrusion et de confiance en l'appareil connecté, notre objectif n'était pas de prévenir l'exécution malveillante mais uniquement de la détecter.

Enfin la communication entre la cible et le moniteur est un canal d'attaque à prendre en compte. Une écoute de ce canal pourrait permettre de récupérer des informations sensibles, celle-ci fournissant un historique complet du flot de contrôle de la cible. De plus, il est indispensable d'éviter de se retrouver dans un scénario de type "attaque de l'intercepteur" [1] (ou *man-in-the-middle* en anglais) qui permettrait à l'attaquant de contrôler la trace envoyée au moniteur permettant ainsi d'attaquer la cible tout en envoyant des traces légitimes. Pour cela, il est indispensable que la communication avec le moniteur soit réalisée via un canal de communication filaire dédié. Pour plus de sécurité ou dans le cas ou il serait impossible de recourir à un canal filaire dédié, une couche de cryptographie pourra être ajoutée afin de sécuriser la communication avec le moniteur, cependant le recours à la cryptographie serait coûteux pour la cible.

Impact sur les performances

En ce qui concerne les performances, EE-CFI permet de déporter l'essentiel du traitement (c'est à dire la vérification) vers un dispositif tiers. La cible se contente d'envoyer des marqueurs qui sont insérés dans son code source vers le moniteur. Le surcoût se

[1] L'attaque de l'intercepteur consiste à relayer secrètement une communication entre deux parties de manière à espionner ou modifier son contenu

limite donc aux quelques instructions nécessaires à la production de la trace et au coût de communication.

Il est à noter que beaucoup d'appareils embarqués permettent d'utiliser le *DMA* ou *direct memory access*. cette solution permet un accès direct entre le matériel responsable de la communication et la mémoire, le processeur indique au matériel qu'il souhaite envoyer des données et le matériel se charge du reste. Cette technique permet de réduire le temps CPU consacré à la communication. Le traitement de la communication peut être exécuté par le matériel de façon asynchrone avec la poursuite de l'exécution du programme.

Enfin il est nécessaire d'évaluer la quantité de données qui seront envoyées afin de s'assurer que le canal de communication ne sera pas surchargé. C'est également pour cette raison qu'il est conseillé d'utiliser un canal de communication dédié afin de ne pas impacter les communications nécessaire au fonctionnement de la cible mais également afin que celles-ci n'impactent pas le dispositif de défense. En effet une surcharge volontaire du canal de communication par l'attaquant pourrait augmenter significativement le délai de transmission de la trace d'exécution et ainsi retarder la détection.

3.2.2 Externalisation : modèle

Le modèle général de EE-CFI est décomposé en trois étapes. Première étape, l'extraction lors de la compilation d'un graphe de flot de contrôle qui nous servira de modèle de référence. Deuxième étape, l'instrumentation du code source afin de produire une trace des transitions dans le flot de contrôle durant d'exécution. Finalement, troisième étape, la trace est envoyée en temps réel au moniteur par la cible et le moniteur vérifie si celle-ci est en accord avec le graphe de flot de contrôle. La figure 3.3 représente le fonctionnement général de la solution.

Graphe de flot de contrôle

Dans le but de spécifier le comportement attendu du programme, EE-CFI utilise l'analyse statique pour extraire un CFG. Chaque appareil embarqué a ses propres points critiques et EE-CFI est conçu pour agir à différents niveaux de granularité allant de la surveillance des appels de fonction, de blocs de base jusqu'à la surveillance d'instructions spécifiques. De ce fait, lors de la création du CFG un identifiant unique est associé à chaque élément (en fonction de la granularité) et le CFG contient toute les transitions directes entre ces éléments comme illustré dans la figure 3.4.

3.2.3 Module d'envoi de trace

Le rôle principal des fonctions prototypées ci-dessus est de récupérer l'identifiant de la fonction et de l'envoyer au moniteur. Les modalités de cet envoi dépendent principalement du matériel ciblé et du matériel utilisé comme moniteur. La question qui se pose alors légitimement est de savoir quel périphérique matériel est susceptible ou non d'être utilisé. Le goulot l'étranglement est ici le volume de données que représente la trace d'exécution à transmettre. C'est donc sur ce point précis que porte notre évaluation afin d'évaluer quel type de matériel pourra être compatible avec notre solution.

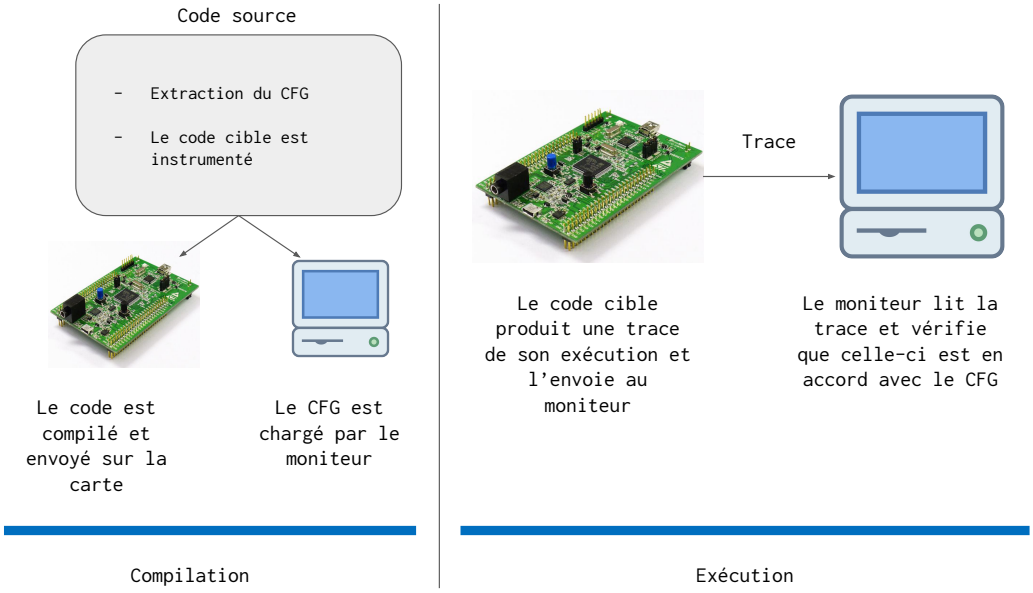

FIGURE 3.3 – Fonctionnement général

Pour nos expérimentations nous avons utilisé un micro-contrôleur STM32F4 comme cible et un ordinateur de bureau sous Linux comme moniteur. Les deux entités communiquaient ensemble via un lien série.

3.2.4 Moniteur

Les traces sont ensuite envoyées au moniteur qui vérifie la cohérence de la trace. Le fonctionnement de cette vérification est détaillée dans le chapitre 4 dans la figure 4.3.

Trace

La trace est composée d'un ensemble de transitions dans le flot de contrôle et, lors de la compilation, du code est inséré avant et après chaque transition. Par exemple lorsque l'on trace les appels de fonction, une trace est produite avant l'appel, à l'entrée dans la fonction, à la sortie de la fonction et après l'appel. De cette manière, tout détournement d'appel ou de retour est détecté immédiatement et sans ambiguïté. La couverture de protection est modulaire, EE-CFI est applicable sur tout ou partie du code dans le but de protéger par exemple uniquement le système d'exploitation ou une fonction critique. Ceci permet de réduire significativement le surcoût d'exécution. La figure 3.5 donne un exemple de trace d'exécution théorique.

```
int pre_calc[100];

int fibbo(int n)
{
    if (n < 100) {
        if (pre_calc[n] == -1){
            pre_calc[n] = (fibo(n-1) + fibo(n-2));
        }
        return precalc[n];
    }
    return fibo(n-1) + fibo(n-2);
}

void init_precalc(){
    pre_calc[0]=0;
    pre_calc[1]=1;
    for(int i=2; i <100; i++){
        pre_calc[i]=-1;
    }
    return;
}

int main()
{
    init_precalc();
    fibo(3);
    return 0;
}
```

```
CFG_main : init_precalc fibo
CFG_fibo : fibo
CFG_init_precalc :
```

FIGURE 3.4 – fibonacci.c : exemple de programme et CFG associé

```
Entree dans main()

Appel de pre_calc()
    Entree dans pre_calc()
    Sortie de pre_calc()
Retour de pre_calc()

Appel de fibo(3)
    Entree dans fibo(3)
    Appel de fibo(2)
        Entree dans fibo(2)
        Sortie de fibo(2)
    Retour de fibo(2)
    Sortie de fibo(3)
Retour de fibo(3)

Sortie de main()
```

FIGURE 3.5 – Exemple de trace d'exécution associée au programme fibonacci.c

Moniteur

Le moniteur lit la trace et vérifie chaque transition. Ce moniteur peut être interne (code injecté, processus séparé), local (périphérique usb, smart card) ou distant (service web, bluetooth, etc..). Le Moniteur lit les transitions contenues dans la trace et regarde dans le CFG si celles-ci sont légitimes. EE-CFI est conçu comme un système de détection d'intrusion c'est à dire que le moniteur se contente de lever une alerte prévenant que le dispositif n'est plus fiable, aucune action directe n'est effectuée sur la cible par le moniteur. Nous avons fait ce choix car étant donné que le matériel n'est pas protégé, il n'y a aucune garantie de pouvoir prévenir une attaque, dans ce cas il devient surtout important de pouvoir identifier si l'appareil est fiable ou non.

CFI complet, orienté et sans état

Étant donné que le traitement de vérification est externalisé et ne pèse pas sur la cible à protéger, nous avons fait le choix d'une politique de CFI précise (ou *fine-grained CFI* en anglais). De ce fait, pour répondre aux problèmes concernant les politiques de CFI énoncées dans le chapitre 2, les choix d'implémentation suivants ont été faits :

CFI complet : Faire du CFI complet revient à vérifier l'intégralité des transitions (pré/post appel et pré/post retour) ce qui garantie une détection rapide d'un détournement du flot de contrôle.

CFI orienté : Un CFG orienté permet quant à elle de protéger le CFI contre l'utilisation de fonctions de distribution utilisées pour faire du *control flow bending* [18] (cf. 2.4.6).

CFI sans état : Finalement le CFI est sans état, ce qui signifie que chaque transition est validée sans tenir compte des transitions précédentes.

3.2.5 Évaluation

Afin de juger de la pertinence de cette approche il est nécessaire d'évaluer sa faisabilité et sa pertinence.

Pour l'évaluation de la faisabilité, la contrainte principale que nous avons est que cette solution doit être déployable sur des systèmes embarqués contraints. Nous devons donc évaluer plusieurs métriques relatives à ces contraintes à savoir : surcoût d'exécution pour le CPU, empreinte mémoire vive, empreinte mémoire du binaire et coût de la communication.

Enfin, pour évaluer la pertinence nous devons nous assurer que le débit de la communication est réaliste par rapport aux possibilités de transmission d'un petit système. Typiquement un petit système embarqué comme un micro-contrôleur peut sortir un débit de l'ordre du mégaoctet par seconde. Nous devons également vérifier que le surcoût d'exécution n'entrave pas le bon fonctionnement du système classique.

3.3 Induction de modèle à partir de traces

La question de l'applicabilité des CFI sur des dispositifs embarqués et l'étude des modalités pour y parvenir est cruciale pour pouvoir envisager l'utilisation des CFI.

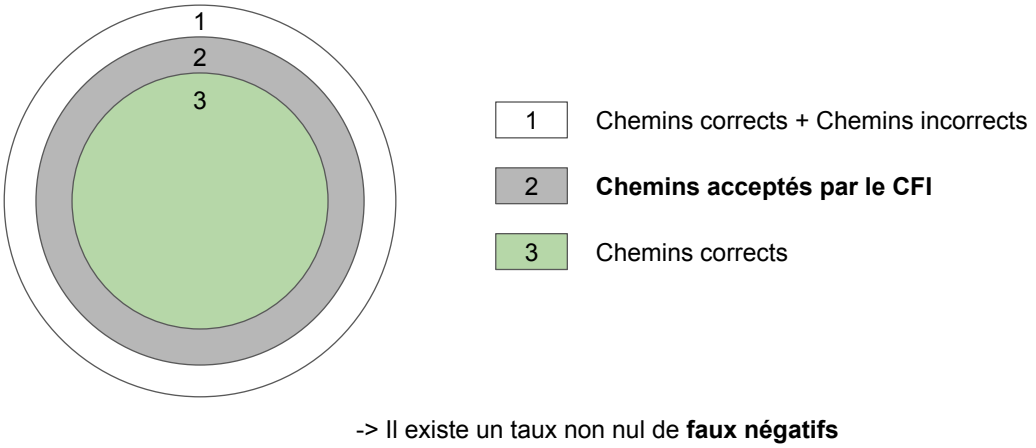

FIGURE 3.6 – L'ensemble des chemins acceptés par le CFI est une extension des chemins corrects

Toutefois le CFI pratiqué actuellement est caractérisé par des vulnérabilités importantes dues à une imprécision du modèle de référence (comme démontré dans la section 2.4.6 du chapitre précédent). Les modèles de CFI les plus précis (*fine grained CFI*), reposent sur l'utilisation d'un graphe de flot de contrôle orienté or celui-ci représente un sur-ensemble par rapport à l'ensemble des exécutions possibles du programme.

3.3.1 Analyse de la couverture de protection

Afin de mieux évaluer la situation il est nécessaire de mieux visualiser la couverture du champs des exécutions autorisées par les modèles de référence classiques.

Pour cette évaluation considérons un modèle de référence ayant les propriétés suivantes :

— Tous les appels directs de fonctions présents dans le code sont identifiés.
— Pour les appels indirects de fonction, une liste de cibles autorisées est identifiée.
— Pour chaque appel, la cible originale du retour est identifiée afin de s'assurer que la fonction retourne bien à son point d'origine.

La figure 3.6 montre l'état initial de la situation. La zone 1 représente l'ensemble des chemins d'exécution, qu'ils soient corrects ou non. La zone 2 représente les chemins acceptés par les solutions de CFI et la zone 3 représente les chemins d'exécution légitimes. Lors d'une attaque, le programme se retrouve alors en situation de chemin incorrect. L'objectif de la défense est donc de différencier précisément un chemin correct (légitime) d'un chemin incorrect (illégitime). Les techniques actuelles de CFI reposent sur une généralisation du modèle réel représenté par le fait que la zone 2 est une extension de la zone 1. Les vulnérabilités du CFI viennent donc de la zone des chemins illégitimes acceptés par les CFI classiques, notre objectif est donc de faire en sorte que les zone 2 et 3 coïncident le plus possible.

3.3.2 Diagnostic de la zone d'imprécision

Dans la suite de mes travaux je me suis penché sur la réduction de cette zone de d'imprécision. Cette zone grise s'explique essentiellement par 4 facteurs : l'imprécision du CFG, la difficulté de prédiction des branchements indirects, l'utilisation abusive des boucles et le code non utilisé.

Imprécision du CFG

Ce que l'on nomme CFG peut prendre en réalité plusieurs formes, allant d'une simple liste de transitions autorisées à un graphe précis de l'enchaînement des appels. Dans le cas de la liste de transitions l'imprécision est évidente, il suffit qu'une fonction soit appelée une fois dans une condition bien particulière pour que celle-ci soit présente dans la liste et puisse ensuite être utilisée à volonté. Mais dans le cas d'un graphe d'enchaînement d'appels il existe également une forme de laxisme due cette fois ci à l'absence de contexte. Les branches conditionnelles sont considérées comme des branches différentes du graphe mais, une fois terminée (i.e. après exécution du "if/else"), l'exécution rejoint alors le même point dans le graphe.

Or dans des conditions réelles d'exécution de nombreux branchements conditionnels sont corrélés, par exemple si plusieurs branchements successifs testent une variable de type "is_admin" et que celle-ci ne peut pas être modifiée entre ces branchements alors il est impossible de se retrouver dans la branche "true" du premier test puis dans la branche "false" du deuxième. De plus, certaines conditions (en particulier les conditions d'erreur) impliquent une rupture du flot de contrôle (retour anticipé de la fonction ou arrêt du programme par exemple). Un cas sain d'exécution ne doit donc pas exécuter le code relatif à ce branchement d'erreur et continuer l'exécution alors que celle-ci aurait du s'arrêter.

Prédiction des branchements indirects

Comme nous l'avons vu dans la section 2.4.5, déterminer avec précision la cible d'un branchement indirect est difficile et les solutions actuelles de CFI se contentent d'une liste plus ou moins précise de transitions probables. Les fonctions superflues de cette liste constituent donc une imprécision.

Utilisation abusive des boucles

la problématique de l'absence de contexte dans les CFG se retrouve également pour l'exécution des boucles. En effet même lorsqu'elles sont détectées, les boucles ne sont pas bornées et une boucle fixe de 4 itérations peut se transformer par une action malveillante en boucle infinie permettant de lire l'intégralité de la mémoire ou d'écrire une quantité arbitraire de données.

Code non utilisé

Lors de la compilation d'un code source vers un binaire, il arrive régulièrement que ce code représente un sur-ensemble du code qui sera effectivement exécuté par la cible.

En effet, il est assez rare que les cas concrets d'utilisation correspondent à 100% du code, certaines fonctions du code ne sont jamais utilisées et certaines parties du code ne sont tout simplement pas atteignables en pratique. On appelle ces sections le code "mort", il comprend par exemple des conditions d'erreur qui ne sont pas atteignables lors d'une exécution saine et qui servent principalement à anticiper une éventuelle erreur de développement.

3.3.3 Retour sur les spécificités de l'embarqué

Afin de proposer une solution à ce problème, il est possible de s'appuyer sur l'un de nos prédicats de départ à savoir que nous nous intéressons à l'informatique embarqué critique. Si l'on dresse la liste des spécificités de ces systèmes alors il possible de voir si de ces spécificités peut émerger une nouvelle approche.

Les spécificités des systèmes embarqués sont les suivantes :
- Ressources limitées.
- Faible surcouche logicielle.
- Matériel peu cher présentant un retard sur l'état de l'art de la sécurité matérielle.
- Difficulté de mise à jour.

La présence de ressources limitées induit une caractéristique : la surface de code à protéger est elle aussi par conséquent limitée. De plus la difficulté (voir l'impossibilité) de mise à jour de ces systèmes critiques pousse également les constructeurs à prévoir un jeu de tests unitaires et fonctionnels beaucoup plus fourni que dans les systèmes classiques qui passent souvent par un cycle de tests utilisateurs : versions alpha et beta et mécanismes de remontée de bug avec parfois des rapports d'erreur détaillés.

3.3.4 Vers un processus d'apprentissage automatique

En résumé les systèmes embarqués possèdent une base de code réduite et des processus de développement qui visent des couvertures de test importantes. Ce contexte semble propice à une observation exhaustive du comportement du programme. De cette manière, en observant uniquement des traces d'exécution légitimes nous pourrions apprendre son fonctionnement réel du programme et ne serions donc pas sujet à la sur-généralisation. Ces traces peuvent être obtenues grâce à l'instrumentation présentée dans la section précédente et une phase de fuzzing de l'application.

Nous proposons donc de mettre en place un mécanisme d'apprentissage automatique capable d'induire un nouveau modèle de référence à partir de traces d'exécutions saines. Cette approche a pour objectif d'améliorer la précision du modèle par rapport aux techniques traditionnelles basées sur les CFG en limitant les transitions possibles uniquement aux transitions observées et en identifiant clairement les cibles d'appels indirects.

3.3.5 Évaluation

L'évaluation de ce modèle repose sur deux aspects. Premièrement, la convergence du modèle, il est indispensable que le mécanisme d'apprentissage puisse converger vers un modèle stable si on lui fournit un nombre raisonnable de traces bien choisies. L'évaluation de la convergence repose sur l'utilisation d'un sous ensemble du jeu de

données à notre disposition pour l'apprentissage du modèle avant de valider ce modèle avec l'ensemble des traces. L'objectif est de déterminer la proportion de trace nécessaire afin d'obtenir un modèle capable de reconnaître l'ensemble des traces. Deuxièmement, la sécurité du modèle, afin de valider la viabilité de cette approche il faudra comparer la précision de notre modèle avec l'existant (graphe de flot de contrôle).

3.4 Synthèse

En résumé, afin de répondre aux problèmes de sécurité des systèmes embarqués, je propose EE-CFI, une solution de CFI externalisée. Cette solution, motivée par un besoin de performance et d'isolation, reposera sur l'externalisation moniteur de surveillance vers un dispositif externe. En complément d'EE-CFI j'évaluerai la possibilité d'utiliser des traces réelles d'exécution dans le but d'améliorer la précision du modèle de référence. Pour évaluer la pertinence de cette démarche, nous évaluerons l'impact de l'externalisation sur la cible afin de valider si celle-ci est effectivement déployable sur des systèmes embarqués. Finalement, pour évaluer la pertinence de l'apprentissage nous proposerons une preuve de concept ainsi qu'une évaluation de celle-ci.

Chapitre 4

Implémentation : externalisation

Ce chapitre détaille l'implémentation de notre proposition pour externaliser la vérification du flot de contrôle. La première partie détaille le développement de la solution et la deuxième partie concerne la mise à place de l'évaluation et les résultats.

4.1 Externalisation : développement

Dans le chapitre 3, nous avons présenté notre modèle d'externalisation. Il repose sur la production d'une trace lors de l'exécution et son envoi pour une vérification par un moniteur externe. L'extraction du modèle de référence et l'instrumentation du code source pour produire une trace d'exécution sont réalisés lors de la compilation. La figure 4.1 est un rappel du fonctionnement général de la solution.

4.1.1 A propos de la chaîne de compilation

Nous avons choisi d'utiliser llvm [60] comme *framework* de compilation et d'instrumentation. LLVM est une infrastructure de compilation modulaire permettant de développer des outils réutilisables d'analyse et d'optimisation de code. Ce choix a été motivé par les raisons suivantes :

Compatibilité : Dans LLVM, l'analyse de code se fait à partir d'un langage intermédiaire appelé *intermediaite representation* ou LLVM-IR [61] qui permet de s'abstraire totalement du langage ainsi que de l'architecture cible.

Outil d'analyse : LLVM est un outil capable d'effectuer des analyses complexes de code comme l'analyse de boucle.

Granularité : LLVM peut opérer à différents niveaux de granularité : instructions, blocs de base, fonctions et même modules (la représentation correspondant à un fichier C ou d'une classe java par exemple). Il est par exemple possible de surveiller les transitions d'un module à un autre ou de restreindre l'accès à un bloc de code particulier. Dans cette étude, nous évaluons les transitions entre fonctions uniquement.

LLVM permet de procéder à l'analyse du code source selon un modèle en 3 temps : initialisation, analyse, finalisation.

Initialisation : La phase d'initialisation permet de récupérer les métadonnées associées au module (classe java ou fichier C par exemple) ainsi que d'effectuer tous les pré traitements nécessaires.

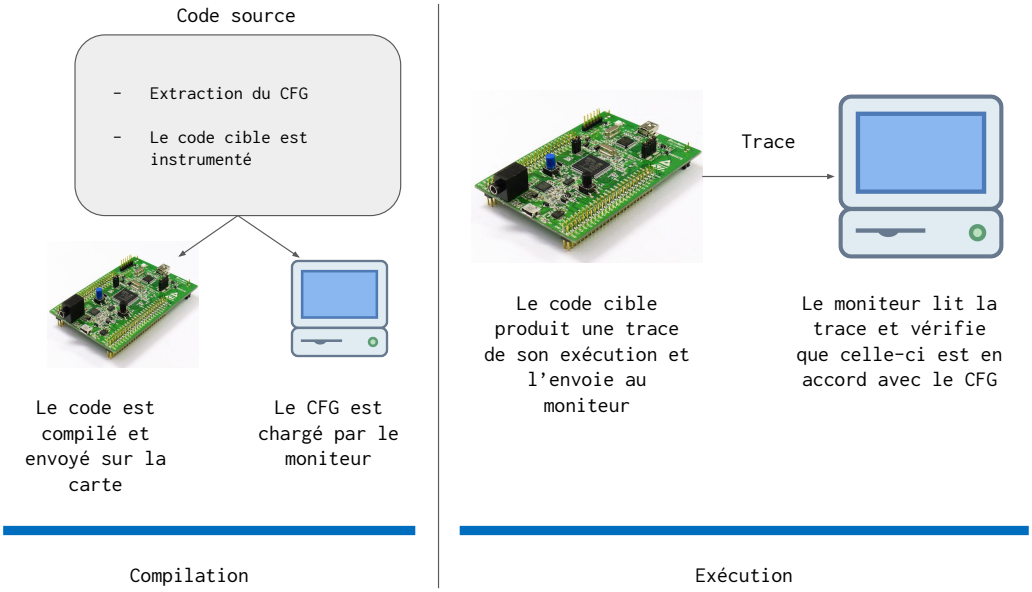

FIGURE 4.1 – Rappel : Fonctionnement général

Traitement : La phase de traitement est exécutée pour chaque fonction du module et donne accès à un itérateur permettant de parcourir le code instruction par instruction.

Finalisation : La phase de finalisation permet d'effectuer les post traitements.

Cet outil est donc adapté pour itérer dans le code afin de détecter les appels de fonction et d'injecter les instructions nécessaires à la génération de trace. Cependant, il n'existait pas encore de module capable d'effectuer les traitements nécessaires à l'analyse et l'instrumentation du code. C'est pourquoi, pour les besoins de cette solution, nous avons développé un nouveau module EE-CFI que nous avons intégré localement à la chaîne de compilation LLVM.

4.1.2 Extraction du modèle de référence

L'extraction de ce modèle est semblable à celle utilisée dans d'autres solutions de CFI [27]. Comme il a été détaillé dans le chapitre 3, afin de vérifier le comportement d'un programme il est nécessaire d'avoir identifié au préalable un modèle de référence (3.1).

EE-CFI utilise un graphe de flot de controle comme modèle de référence, ce modèle enregistre pour chaque fonction la liste des fonctions susceptibles d'êtres appelées. Un graphe orienté est une structure de donnée composée de nœuds reliés entre eux par des arcs. Le graphe de flot de contrôle est obtenu en associant à chaque fonction un noeud et en reliant celui-ci par des arcs vers toutes les transitions légitimes identifiées précédemment. Ici l'aspect orienté de graphe de flot de contrôle vient du fait qu'il y a une différentiation entre fonction appelante et fonction appelée. Les arcs sont orientés et une transition légitime de A vers B n'implique pas une transition légitime de B vers A.

Le module LLVM EE-CFI parcours donc les instructions présentes dans chaque fonction à la recherche d'appels de fonction. Ces appels sont représentés en LLVM-IR par

deux types d'instruction qui correspondent respectivement aux appels directs et aux appels indirects. La fonction isa<TYPE>, qui permet de tester le type d'une instruction, est utilisée pour chaque instruction afin de savoir si celle-ci est un appel de fonction direct (CallInst) ou indirect (InvokeInst). Une fois un appel détecté, le module ajoute un nouveau noeud dans le CFG et une transition orientée entre la fonction appelante et la fonction appelée. Dans le cas d'un appel indirect, la fonction appelée n'est pas clairement identifiée, dans ce cas le module ajoute une transition vers un état *wildcard* pouvant prendre comme valeur toutes les fonctions identifiées du programme.

Afin d'établir une interface pour la communication entre le moniteur et la cible, un identifiant est attribué à chaque fonction. Cet identifiant est unique pour chaque fonction et sera utilisé dans le CFG et dans la trace. Le format texte n'étant pas optimal pour la communication tant du point de vue de la sécurité où il offre une trop grand visibilité que du point de vue des performances où il occupe une taille en mémoire importante, un nouvel identifiant plus petit (basé sur l'identifiant donné automatiquement par LLVM) est attribué à chaque fonction.

4.1.3 Instrumentation de la trace

Afin que le code fournisse une trace de son exécution, des instructions doivent être ajoutées au code. Ces instructions ont pour but de produire une trace chargée d'indiquer chaque branchement du flot de contrôle. La trace se compose de l'ensemble des appels et retours de fonction ainsi que des entrées et sorties de ces fonctions.

Par exemple si une fonction A appelle une fonction B, la trace attendue est :

`Entrée A > Appel B > Entrée B > Sortie B > retour B > Sortie A`

Afin de réaliser cette trace il faut donc injecter du code à l'entrée et à chaque retour de fonction (i.e. avant chaque instruction *return*), ainsi que avant et après chaque appel.

Pour faire cela il existe deux méthodes : la première est d'ajouter une à une toutes les instructions nécessaires pour réaliser ce que l'on souhaite faire. Cette méthode, plus optimisée en terme de performance est cependant extrêmement fastidieuse, chaque variable, chaque constante doit être préalablement déclarée pour être correctement inséré au modèle généré par LLVM. De plus l'augmentation de la taille du binaire final (après instrumentation du code) est très importante vu que le code injecté n'est pas factorisé dans des fonctions.

C'est pour cela que j'ai choisi la deuxième solution qui est d'insérer uniquement un appel à une fonction externe et de lier à la compilation une librairie C contenant toutes les fonctions que l'on souhaite insérer. De cette façon il est possible d'effectuer toutes les modifications directement en C dans la librairie sans avoir à réécrire la passe d'instrumentation et instrumenter le code à nouveau. Cette méthode offre également une plus grande modularité dans le sens ou les fonctions insérée sont des interfaces et il est possible de posséder plusieurs versions de la librairie C effectuant des traitements différents.

4 Fonctions sont nécessaires pour effectuer une vérification complète :

— int llvm_cfi_in(int function);
— int llvm_cfi_out(int function);
— int llvm_cfi_callf(int function);
— int llvm_cfi_returnf(int function);

```
void fonction_a(){
        //...
        fonction_b();
        //...
    return;
}
```

```
void fonction_a(){
        llvm_cfi_in(getid(a));
        //...
        llvm_cfi_callf(getid(b));
        fonction_b();
        llvm_cfi_returnf(getid(b));
        //...
        llvm_cfi_out(getid(a));
    return;
}
```

FIGURE 4.2 – Exemple d'instrumentation d'une fonction

Ces fonctions ont pour paramètres l'identifiant de la fonction concernée et seront appelées respectivement à l'entrée de chaque fonction, avant le retour de chaque fonction, avant chaque appel et après chaque appel comme illustré dans la figure 4.2.

4.1.4 Module d'envoi de trace

Le rôle principal des fonctions prototypées ci-dessus est de récupérer l'identifiant de la fonction et de l'envoyer au moniteur. Les modalités de cet envoi dépendent principalement du matériel cible et du matériel utilisé comme moniteur. La question qui se pose alors légitimement est de savoir quel périphérique matériel est susceptible ou non d'être utiliser. Le goulot l'étranglement est ici le volume de donné que représente la trace d'exécution à transmettre. C'est donc sur ce point précis que porte notre évaluation afin d'évaluer quel type de matériel pourra être compatible avec notre solution.

Pour nos expérimentations, nous avons utilisé un micro-contrôleur STM32F4 comme cible et un ordinateur de bureau sous Linux comme moniteur. Les deux entités communiquaient ensembles via un lien série.

4.1.5 Moniteur

Les traces sont ensuite envoyées au moniteur qui vérifie la cohérence de la trace. Le fonctionnement de cette vérification est détaillé dans la figure 4.3.

```
toReturn=""
toCall=""
stack=[]
first = 1

for item in trace :

# Initialisation : la fonction d'entree ne necessite pas
# de verifications d'appel
        if (item.operation == In) and first :
        first = 0
        stack.append(item.function)

# On verifie a l'entree d'une fonction qu'on arrive bien dans
# la fonction prevue, si c'est le cas on reinitialise toCall
        elif (item.operation == In) and not first :
                if item.function != toCall :
                        print ('corrupted call')
                else :
                        toCall=""
                stack.append(item.function)

# A la sortie d'une fonction on enregistre la fonction que
# l'on quitte
        elif (item.operation == Out) :
                toReturn = item.function

# Au retour d'une fonction on verifie qu'on est bien revenu
# au bon endroit; si c'est le cas on reinitialise toReturn
        elif (item.operation == Returnf) :
                stack.pop()
                if item.function != toReturn :
                        print ('corrupted return')
                else :
                        toReturn=""

# Lors d'un call on enregistre la fonction que l'on appelle
# Ensuite on verifie que la transition est legitime
        elif (item.operation == Callf)
                toCall=item.function
                if toCall not in cfg[stack.peek()] :
                        print ('unregistered call')

# Sauf si ils sont en attente de validation, les valeurs de
# toReturn et toCall doivent avoir ete reinitialisees, sinon,
# le flot de controle a ete detourne
        if (item.operation != Out) and (toReturn != "") :
                print ('corrupted return')
        if (item.operation != Callf) and (toCall != "") :
                print ('corrupted call')
```

FIGURE 4.3 – Fonctionnement de l'algorithme de vérification

En plus de vérifier que la trace est bien formée, le moniteur valide également les transitions en les comparant aux transitions autorisées par CFG.

Ce moniteur étant temporaire il n'a pas été évalué, cependant celui-ci reprend le principe de base du CFI à savoir une vérification stricte des appels et retours de fonction et donc offre les mêmes garanties de sécurité. De plus dans cette première implémentation le problème de l'intégrité des appels indirects n'est pas abordé, il sera abordé dans le chapitre 5.

4.2 Externalisation : évaluation

Afin de valider la pertinence de cette approche, une phase d'évaluation visant à mesurer la faisabilité et la pertinence de la démarche a été mise en place. Lors de cette phase, l'impact de la solution sur une cible embarquée est évalué afin de vérifier que la solution est compatible avec les contraintes des systèmes embarqués. L'évaluation se base sur trois critères : surcoût d'exécution pour la cible, augmentation du volume de code et coût de la communication avec le moniteur.

4.2.1 Spécifications matérielles

En réponse aux préoccupations de performance et de sécurité, la partie vérification d'EE-CFI est externalisée. Cette externalisation réduit le surcoût sur le processeur et protège d'une attaque ciblant directement la mémoire de la solution de CFI en isolant physiquement le moniteur de vérification et la cible. Le matériel utilisé lors des tests est la carte STM32F407VG [95] qui est un exemple typique de micro-contrôleur avec une faible sécurité matérielle et une capacité de mémoire et de calcul limitée. Celle-ci est équipé d'un processeur ARM Cortex-M4 fonctionnant à 168Mhz, d'une mémoire flash de 1Mo, de 192Kb de SRAM et embarque une MPU. Cette carte (cible à défendre) communique avec un ordinateur (moniteur) via un lien série cadencé à 921600 Bauds soit un débit d'environs 115ko/s.

4.2.2 Spécifications logicielles

Nous avons choisi d'utiliser le système d'exploitation FreeRTOS [88] (version 9.0.0). Ce système open source est conçu pour les systèmes critiques et propose un ordonnancement en temps réel. Sa conception est volontairement minimaliste pour qu'il puisse être installé sur des petits systèmes. Le kit minimal ne compte que quelques fonctions pour gérer les tâches et la mémoire. Nous avons choisi ce système pour notre évaluation car il est déstiné aux petits systèmes embarqués critiques. De plus, ce système n'incluant pas de mécanismes de protection contre les attaques détournant le flot de contrôle, il représente un cas d'application typique pour notre solution.

4.2.3 Benchmarks

Lors de l'instrumentation, des marqueurs sont insérés au début et à la fin de chaque fonction ainsi qu'avant et après chaque appel. De ce fait, un programme contenant une seule fonction et donc aucun appel aura un surcoût d'exécution quasi nul. Par

contraposée, un programme contenant un montant important d'appels de fonction réalisant des petites taches présente un surcoût d'exécution important.

Pour les expérimentations nous avons considéré 2 micro benchmarks :

Fibonacci (récursif) : Pire scénario, beaucoup d'appels de fonction réalisant très peu de calculs.

Fibonacci (itératif) : Meilleur scénario, presque aucun appel de fonction.

Les paramètres de test utilisés pour ces scénarios sont fibonacci(20) calculé 50 fois pour la version récursive et fibonacci(2000) calculé 1000 fois pour la version itérative.

Nous avons également utilisé les benchmarks suivants :

Dhrystone [105] : Benchmark utilisé couramment dans les systèmes embarqués.

AES encryption/decryption : Ce benchmark effectue plusieurs opérations avec la méthode cryptographique AES [29]. Il représente un cas normal d'utilisation avec des calculs complexes répartis dans plusieurs fonctions.

FreeRTOS : Afin d'évaluer l'impact de notre solution sur un système dexploitation, un second résultat est présenté pour tous les benchmarks décrits précédemment en instrumentant cette fois ci FreeRTOS plutôt que le code du benchmark lui même.

4.2.4 Protocole expérimental

Ces expérimentations visent à évaluer la capacité de notre solution à fonctionner sur des appareils embarqués contraints en mémoire et capacité de calcul. Nous avons considéré les 3 métriques suivantes :

1. Le surcoût CPU sur la cible. Nous évaluons le nombre de cycles CPU supplémentaires nécessaires à notre solution pour avoir une estimation du pourcentage d'augmentation de l'utilisation du CPU.
2. La quantité de données échangées entre la cible et le moniteur. Le débit en kilo-octet par seconde est évalué pour vérifier qu'il est compatible avec les canaux de communications présents dans les systèmes embarqués.
3. Le surcoût sur la taille du binaire à charger dans le micro-contrôleur est évalué et comparé à la taille initiale de la cible. Les systèmes embarqués étant contraints en mémoire, la solution doit limiter autant que possible son impact sur la mémoire.

Nous avons évalué ces métriques sur les 4 scénarios suivants :

Base : Programme non ordonnancé, sans instrumentation. Ceci est le scénario de référence.

Benchmark : Programme non ordonnancé, avec instrumentation du benchmark. Ce scénario présente le surcoût total de la solution, tout le code exécuté est instrumenté.

Ordonnancé : Programme ordonnancé sans instrumentation. Dans le but d'évaluer l'impact de l'instrumentation de FreeRTOS, ce benchmark sert de référence. Une tache contenant le benchmark est créée ainsi qu'une tache vide pour forcer l'ordonnancement (boucle while vide), l'ordonnanceur est ensuite lancé.

FreeRTOS : Programme ordonnancé avec un FreeRTOS instrumenté. Ce scénario permet de voir l'impact de la solution sur FreeRTOS.

4.2.5 Résultats

Surcoût CPU

Pour évaluer le surcoût CPU, seule la production de la trace en mémoire est considérée. Le coût de communication n'a pas été pris en compte étant donné qu'il dépend principalement de l'implémentation choisie et du matériel. De plus une implémentation optimale de la communication repose sur la configuration d'un DMA. Or l'utilisation d'un DMA par le lien série n'a qu'un impact marginal sur les performances du CPU. Les résultats ci-dessous montrent les temps d'exécution en nombre de cycles CPU, ainsi que le surcoût correspondant pour les versions instrumentées (exprimé en %).

Comme attendu, le tableau 4.1 montre que le programme entièrement linéaire présente un surcoût très faible tandis que le programme récursif présente un surcoût très important. Le cas de FreeRTOS présente quant à lui un surcoût assez faible, aux alentours de 3% ce qui semble acceptable. Ces résultats indiquent également qu'une protection totale du dispositif dans des cas normaux d'utilisation (benchmarks dhrystone et aes) présentent un surcoût d'exécution important (+1306% et +354%). On peut en conclure que la sécurisation d'un appareil embarqué avec un CFI externalisé peut nécessiter de restreindre la partie du code surveillée aux fonctions critiques du système.

Volume de donnée liées à la communication

Le critère principal pour évaluer la charge entrées-sorties entre la cible et son moniteur est le volume de données sortant. Pour faire cela, nous avons compté le nombre de traces (c'est à dire le nombre de transitions dans le flot de contrôle) devant être envoyé. Le tableau 2 montre le nombre de traces envoyées par secondes ainsi que le coût de communication (en kilo-octets par seconde) correspondant.

Les résultats du tableau 4.2 sont directement liés au surcoût CPU, ce qui s'explique facilement par le fait que tout ce que fait l'instrumentation n'est que la génération de trace. On constate également que dans le cas le plus défavorable le volume de données envoyé est inférieur au méga-octet par seconde ce qui le rend incompatible avec une connexion UART mais compatible avec les canaux de communication Ethernet et USB 2.0.

Taille des binaires

L'instrumentation du code source génère une augmentation de la taille du binaire, lié cette fois au nombre de fonctions et aux nombres d'appels différents de fonction. Cette augmentation dépend donc de la taille du code source initial et de la factorisation du code (nombre de fonctions et nombre d'appels distincts à celles-ci). Il faut noter que dans tous les scénarios la taille du binaire comprend : FreeRTOS, des librairies relatives à la carte STM32F4 et un ensemble de librairies annexes. Le tableau 3 montre la taille du binaire (en octet) pour chacun des scénarios.

TABLE 4.1 – Cycles CPU pour chaque benchmark (en millions de cycles)

	Fibonacci (récursif)	Fibonacci (itératif)	Dhrystone	AES
Base	46,90	76,02	103,12	157,77
Benchmark	1967,61 (+4095%)	77,89 (+2.45%)	1450,34 (+1306%)	716,48 (+354%)
Ordonnancé	82,35	132,29	183,92	263,16
FreeRTOS	85,00 (+3.21%)	136,48 (+3.16%)	189,47 (+3.01%)	272,42 (+3.51%)

TABLE 4.2 – Volume de données envoyées au moniteur en nombre de traces (en milliers) par seconde

	Fibonacci (récursif)	Fibonacci (itératif)	Dhrystone	AES
Base	0	0	0	0
Benchmark	356.64 (713.28ko/s)	8.44 (16.88ko/s)	245.86 (491.72ko/s)	109.92 (219.84ko/s)
Ordonnancé	0	0	0	0
FreeRTOS	5.68 (11.36ko/s)	4.03 (8.06ko/s)	4.90 (9.80ko/s)	4.20 (8.40ko/s)

TABLE 4.3 – Taille des binaires (en octets)

	Fibonacci (récursif)	Fibonacci (itératif)	Dhrystone	AES
Base	74416	74416	76508	79820
Benchmark	74900 (+0.65%)	74820 (+0.54%)	78700 (+2.87%)	82404 (+3.24%)
Ordonnancé	74524	74524	76616	79920
FreeRTOS	103824 (+39.32%)	103824 (+39.32%)	105912 (+38.24%)	109216 (+36.66%)

Le tableau 4.3 montre qu'instrumenter les algorithmes n'impacte que très peu la taille du binaire final (maximum +3%), ceci s'explique par le fait que le code relatif aux algorithmes représente un petit volume de donnée comparé à FreeRTOS et aux autres librairies. Instrumenter FreeRTOS en revanche produit une augmentation proche de 40% de la taille du binaire final. Cette augmentation semble acceptable car elle conserve la taille du binaire dans le même ordre de grandeur.

4.2.6 A propos des résultats

Les résultats présentés ci dessus confirment que EE-CFI peut être utilisé en l'état dans la plupart les systèmes embarqués si l'on peut limiter la protection aux portions critiques du code. Cependant de bonnes pratiques sont nécessaires lors de sa mise en place de la solution afin de minimiser le surcoût. Ces résultats peuvent être améliorés en optimisant le code instrumenté dans le but de réduire le surcoût CPU et/ou l'augmentation de la taille des binaires. La taille de la trace générée étant raisonnable, nous pensons donc que si l'on parvient à optimiser suffisamment le code instrumenté alors la solution pourrait même être envisagée pour une surveillance complète du dispositif au prix d'un sacrifice de performance.

4.2.7 Perspectives

Certaines opportunités de la différentiation cible/moniteur sont encore à explorer. Tout d'abord, si l'appareil cible ne peut pas forcément être mis à jour, il est possible de mettre à jour le moniteur, ces mises à jour pourraient permettre une meilleure détection d'attaque en utilisant les traces d'attaques passées afin de détecter au plus vite une compromission.

Le moniteur pourrait également être utilisé pour mutualiser la défense de plusieurs cibles. Par exemple, si l'on s'intéresse à la surveillance de plusieurs capteurs dans un système industriel, il est possible de corréler les données fournies par les différents capteurs et de les analyser les unes par rapport aux autres. Si deux capteurs surveillent la même donnée et qu'ils ont des comportements complètement différents alors il est probable que le système soit défaillant ou qu'il ait été compromis.

Enfin le moniteur peut avoir un rôle dans le diagnostique de crash ou de comportement anormal. Si celui ci est connecté vers l'extérieur il peut permettre une analyse de défaillance à distance. Il peut également être utilisé à des fins d'optimisation pour identifier les sections de codes les plus exécutées ou les plus lentes.

4.3 Synthèse

Dans ce chapitre nous avons détaillé l'implémentation d'EE-CFI, une solution externalisée de CFI dédiée aux appareils embarqués. Cette solution repose sur l'utilisation de la chaîne de compilation LLVM pour extraire un CFG servant de modèle de référence et pour instrumenter le code afin que celui-ci produise une trace de son exécution. Nous avons évalué cette solution à travers différents scénarios dont l'instrumentation du système temps réel FreeRTOS. Lors de cette évaluation nous avons évalué le surcoût d'exécution pour le CPU, le volume de données transitant entre la cible et le moniteur ainsi que l'augmentation de la taille du binaire. Cette évaluation a montré des résultats

pour la surveillance complète de FreeRTOS qui sont en accord avec les contraintes des systèmes embarqués. En revanche, le surcoût d'exécution CPU induit par la surveillance intégrale du code embarqué étant pour l'instant trop élevés, nous en avons déduit qu'en l'état actuel de la solution il était nécessaire de restreindre la surface de code à surveiller afin de limiter l'impact de la solution sur le CPU. Cependant le coût de communication et l'augmentation de la taille du binaire restant dans le domaine de l'acceptable pour un système embarqué nous pensons qu'avec des améliorations de performance cette solution pourrait être utilisée dans une surveillance complète d'un dispositif embarqué.

Chapitre 5

Implémentation : apprentissage

L'exécution d'un programme informatique n'est pas un enchaînement arbitraire d'instructions, l'exécution de ces instructions suit une logique induite par le code source et ce même si l'exécution dépend de données externes (informations fournies par l'utilisateur, données provenant de capteurs etc). Par exemple dans le cas d'un interpréteur de commande, son exécution est certes entièrement dépendante des entrées de l'utilisateur, néanmoins son comportement est entièrement déterministe et il peut donc par conséquent être modélisé : initialisation, affichage du *prompt*, affichage de la saisie utilisateur, *parsing* de la commande, exécution de la commande, affichage du résultat. Ces étapes constituent la logique interne de l'interpréteur de commande et elles sont conçues pour être présentes dans cet ordre précis. Si une commande est exécutée avant la saisie utilisateur ou si l'affichage du résultat intervient avant l'exécution de la commande alors le comportement n'est plus celui attendu.

Afin de différencier un comportement légitime (le comportement réel est conforme au comportement attendu) d'un comportement illégitime (le comportement réel n'est pas celui attendu) il est nécessaire de pouvoir définir le plus précisément possible quel est le comportement attendu. Dans le chapitre 3, nous avons défini que la représentation de ce comportement attendu était appelée "modèle de référence". Ce modèle de référence peut être obtenu par prédiction : le comportement attendu est prédit avant l'exécution à partir de spécifications, du code source ou du binaire. A contrario, il est possible d'obtenir le modèle de référence par déduction : le comportement attendu est déduit après (ou pendant) l'exécution et se base sur l'observation du comportement réel du programme. De plus, dans les chapitres précédents il a été vu que les solutions de CFI actuelles reposent sur des modèles références obtenus analyse de code, et donc par prédiction, qui comportent de nombreuses imprécisions conceptuelles donnant suffisamment de marge de manœuvre à un attaquant grande pour lui permettre d'exploiter les vulnérabilités d'un programme. Dans ce chapitre, nous proposons une implémentation d'un mécanisme déductif de création du modèle de référence dans le but de réduire cette marge de manoeuvre d'attaque.

5.1 Méthodologie Expérimentale

5.1.1 Processus de création du modèle de référence

L'objectif de cette approche est de mettre en place une solution de déduction du comportement attendu, pour cela le processus de création du modèle de référence doit être modifié. Dans le chapitre 4, ce modèle était créé lors de la compilation (cf figure

3.3 du chapitre 3) en construisant une liste de transitions autorisées par analyse du code source, c'était donc un mécanisme de prédiction du comportement.

Le processus de déduction de comportement que nous avons mis en place utilise l'observation d'exécutions réelles et se compose de trois phases : collecte, apprentissage et contrôle.

Collecte : Afin de disposer d'un jeu de données d'entraînement permettant d'apprendre le comportement du programme, le processus de déduction du comportement repose sur une phase de collecte qui consiste à agréger un certain nombre de traces d'exécutions qui serviront de base d'apprentissage. Cette collecte peut s'effectuer dans un environnement contrôlé avant la mise en production ou alors se dérouler en parallèle de celle-ci.

Apprentissage : Une fois que suffisamment de traces ont été accumulées, la phase d'apprentissage réalise une synthèse à partir de l'ensemble ou d'une portion de ces traces, résultant en la création d'un modèle représentant le comportement attendu du programme.

Contrôle : Finalement, le nouveau modèle est confronté à l'ensemble des traces afin de vérifier si il a correctement appris le comportement du programme.

La figure 5.1 montre comment ce nouveau processus de création du modèle de référence s'inscrit dans le processus global de sécurité. On y retrouve les phases décrites précédemment :

A Compilation

B+C Collecte

D Apprentissage

E Contrôle

F+G Exécution

Couverture de code et couverture d'état

Afin de déterminer la méthode à utiliser lors de la phase d'apprentissage, il est nécessaire de réaliser une nouvelle évaluation de la couverture des exécutions possibles dans le cas d'un modèle déduit des exécutions.

Lors de l'extraction de modèle par prédiction il n'est pas pris en compte la réalité de l'exécution, ainsi seul le code source et ses différentes représentations (Code source, LLVM IR, Code binaire) sont pris en compte pour la création du modèle. Afin d'évaluer cette approche, il est possible de mesurer la proportion de code analysée, on parle alors de couverture de code.

Or comme il a été indiqué dans le chapitre 3, l'analyse du code source donne lieu à un certain nombre d'imprécisions : cibles de branchements indirects, branchements conditionnels corrélés, boucles et code non utilisé. Ces imprécisions sont présentes malgré le fait que l'intégralité du code soit analysé et donc que la couverture de code soit de 100%. Afin d'identifier plus précisément la cause de ces imprécisions il est nécessaire d'introduire une nouvelle notion complémentaire à la couverture de code : la couverture d'état.

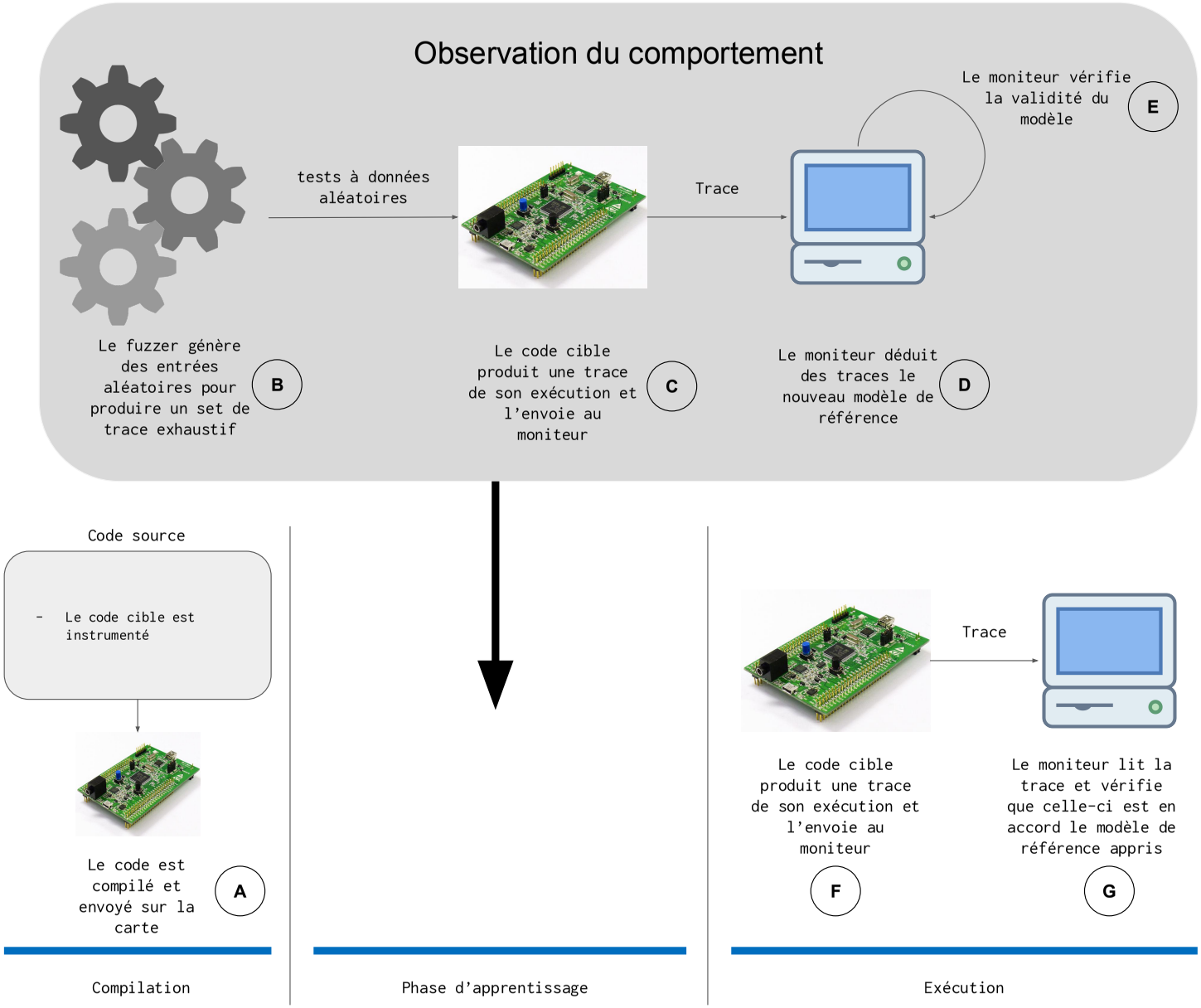

FIGURE 5.1 – Fonctionnement de la création du modèle de référence par apprentissage

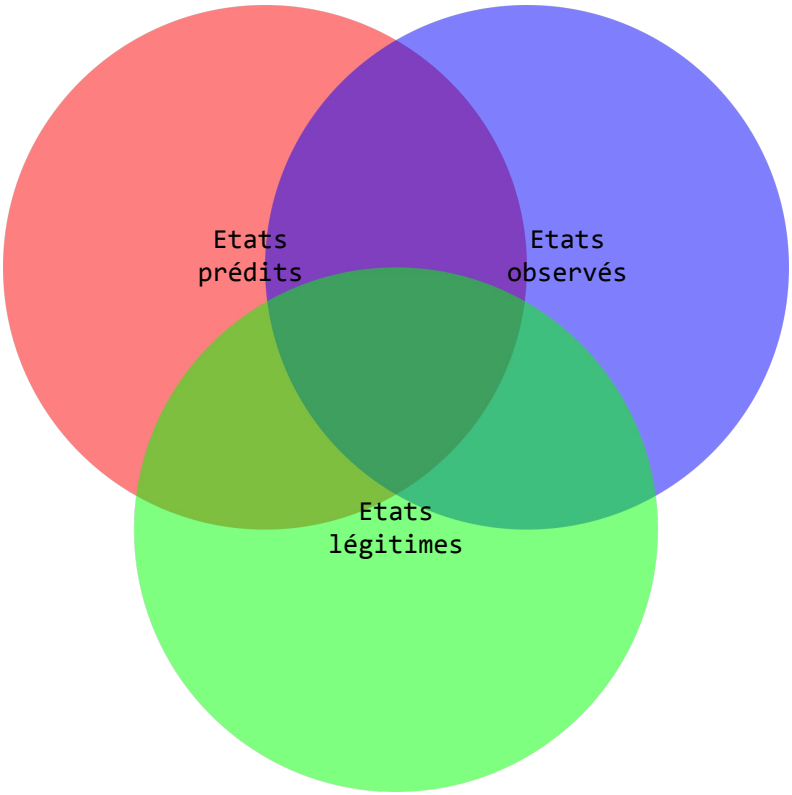

FIGURE 5.2 – Évaluation de la couverture d'état

Un état d'un programme représente à la fois une position dans le code (i.e. une valeur du compteur ordinal) ainsi que la valeur de chacune des variables du programme (i.e. la valeur de l'intégralité de la pile d'exécution). L'ensemble des états d'un programme représente donc l'ensemble des configurations possibles de la pile d'exécution pour chaque valeur du compteur ordinal. Ainsi une couverture de code de 100% (et donc une couverture de toutes les valeurs possible du compteur ordinal) n'implique pas une couverture de 100% des états du programme. De plus, parmi la liste de tous les états théoriques d'un programme, beaucoup ne sont pas atteignables en pratique.

Dans la suite de ces travaux, on distinguera donc les états théoriques du programme (i.e. atteignables et non-atteignables) des états pratiques (i.e. atteints). La couverture d'état d'un programme est définie comme étant la proportion d'états pris en considération dans le modèle de référence par rapport à l'ensemble des états pratiques/atteints du programme. Les états non-atteignables ne faisant pas partie de la réalité d'exécution du programme, il sont considérés comme une source d'imprécision. Parmi ces états non-atteignables, on retrouve donc : des cibles incorrectes de branchements indirects, des états inatteignables de boucles ou de branchements conditionnels corrélés ainsi que le code mort.

La figure 5.2 positionne les ensembles des état déduits et des états prédits par rapport à l'ensemble des états légitimes.

États légitimes : Représente l'ensemble des états pratiques du programme, qui peuvent êtres légitimement atteints lors de l'exécution.

États prédits : Représente l'ensemble des états acceptés par les modèles de références prédits qui est un sous ensemble des états théoriques du programme.

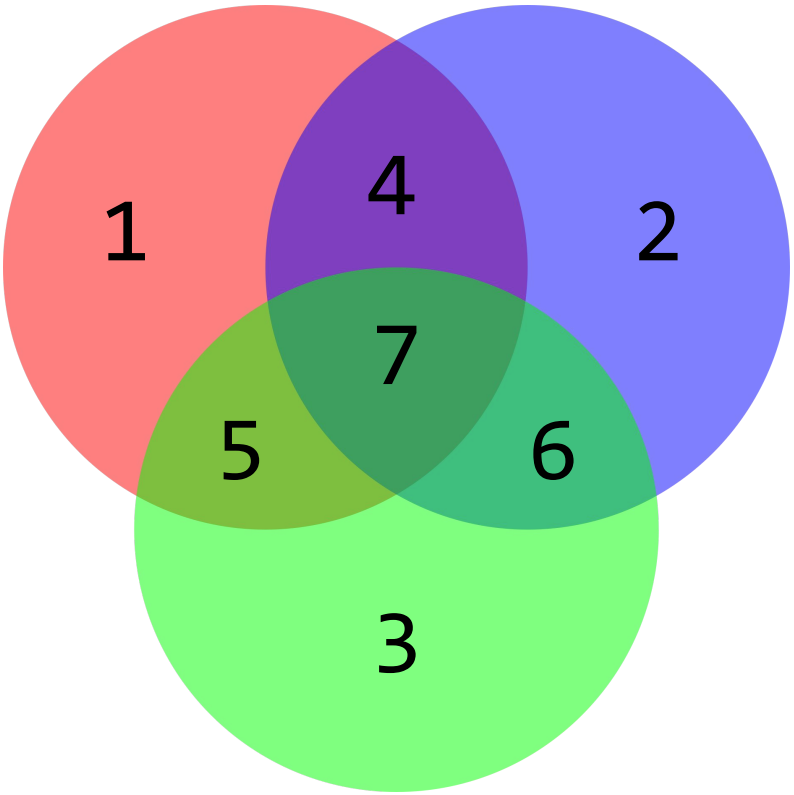

FIGURE 5.3 – Détail de la couverture d'état

États déduits : Représente l'ensemble des états acceptés par les modèles de référence déduits. Cet ensemble est également un sous ensemble des états théoriques du programme.

Comme le montre la figure 5.2 ces différents ensembles ne sont pas identiques, les modèles prédits et déduits incluent donc des états inatteignables et excluent des états atteints. Si l'on s'intéresse aux intersections entre ces ensembles, il est possible de mettre en évidence les sources de cette imprécision. La figure 5.3 reprend les différentes zones identifiables.

1 : États prédits inatteignables. Ce sous-ensemble comprend principalement le code non utilisé et les cibles superflues d'appels indirects.

2 : États déduits inatteignables. Ce sous-ensemble comprend d'éventuels états qui auraient étés atteints lors d'une phase de collecte en pré-production mais qui ne seraient plus atteignables une fois l'appareil mis en production.

5+7 : Etats prédits atteints.

6+7 : Etats déduits atteints.

4+7 : Généralisations. Ces états inatteignables acceptés par les deux types de modèle représentent les généralisations éventuelles effectuées par ces modèles, les généralisations de la zone 4 étant considérées comme des sur-généralisations du modèle.

5 : Etats atteints non observés. Ce sous-ensemble représente les potentiels états atteints en pratique qui n'auraient pas étés observés lors de la phase de collecte ni généralisés lors de la phase d'apprentissage et qui seraient donc absents du modèle de référence déduit.

6 : Etats atteints non prédits. Ces états peuvent exister lorsque l'analyse des cibles potentielles de sauts indirects n'est pas correctement réalisée et que celle-ci interdit des transitions

3 : États légitimes atteints non prédits et non observés. Ces états marginaux peuvent se produire dans de rares cas lors d'un appel indirect.

L'objectif est donc de créer un modèle de référence qui ait une couverture d'état approchant les 100%, c'est à dire que la phase d'apprentissage doit généraliser tous les états légitimes qui n'ont pas étés vu lors de la phase de collecte sans inclure d'états illégitimes. Afin de déterminer la méthode la plus appropriée pour créer un modèle de référence, il est nécessaire de procéder à une évaluation de l'ordre de grandeur du nombre d'état d'un programme.

Considérons la fonction d'addition :

```
long add(unsigned a, unsigned b){
        return a+b;
}
```

Une fois compilée celle-ci produit le code assembleur suivant :

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
mov     eax, eax
pop     rbp
ret
```

Un calcul naïf pour ce programme de 10 instructions manipulant deux entiers positifs (de 0 à 4,294,967,295) donnerait déjà un nombre de combinaisons de $10 * 2^{32} * 2^{32}$ soit environ $1,8 * 10^{20}$ combinaisons. Même sans se lancer dans des calculs complexes, il est évident que le nombre de combinaisons possibles des états atteignables d'un programme (l'ensemble des valeurs atteignables du compteur ordinale multiplié par l'ensemble des combinaisons de valeurs possibles de la pile d'exécution) explose lorsque l'on parle d'un binaire comprenant plusieurs milliers d'instructions ainsi que plusieurs milliers d'octets de données variables dans la pile.

De ce fait, il est évident que chercher à créer un modèle en observant un à un tous les états pratiques n'est pas envisageable. De plus, comme il a été pointé précédemment, l'utilisation d'un modèle purement prédictif repose nécessairement sur une généralisation qui induit beaucoup d'imprécisions. Il est donc nécessaire de faire un compromis entre la généralisation des états théoriques et l'observation des états pratiques. Pour résoudre de façon automatisée ce problème, nous proposons de recourir à un algorithme d'apprentissage automatique capable d'apprendre à partir d'un ensemble d'exemples (énumération d'états pratiques) le comportement général du programme. L'algorithme d'apprentissage aura alors pour fonction de faire des compromis afin de généraliser le modèle suffisamment pour pouvoir reconnaître des états pratiques qu'il n'aura jamais vu tout en rejetant les états inatteignables.

```
void loop_function(int iter){
        init();
        for ( int i = 0; i < iter; i++){
            do_work();
    }
    finalize();
}
```

FIGURE 5.4 – Code source d'une boucle simple

```
loop_function(1);
// la trace produite est de la forme :
loop_function:do_work:loop_function

loop_function(4);
// la trace produite est de la forme :
loop_function:do_work:do_work:do_work:do_work:loop_function
```

FIGURE 5.5 – Exemples de traces pour différentes itérations

5.1.2 Entre non-convergence et sur-généralisation

Afin d'illustrer la problématique de compromis dans un algorithme d'apprentissage, intéressons-nous au cas des boucles. Lorsqu'un programme présentant une boucle s'exécute (comme dans l'exemple 5.4), le code instrumenté produit une trace d'exécution qui est une représentation du programme déroulé, c'est à dire que la boucle n'y apparaît plus et à sa place on peut voir une succession de symboles ou de séquence de symboles.

Ainsi deux exécutions ne différant que par le nombre d'itérations de boucles présenteront deux traces différentes comme le montre l'exemple illustré dans la figure 5.5.

Un automate absolument précis devrait enregistrer toutes les itérations possible d'une boucle et créerait donc, pour chaque boucle et pour chaque nombre d'itérations, un nouveau chemin. De ce fait, lorsque plusieurs boucles indexées par exemple sur un uint32 s'enchaînent le nombre de chemins à créer devient vite trop important (pour rappel leur nombre de combinaisons pour deux entiers positifs codés sur 32 bits est de $1,8*10^{20}$). On se retrouve alors dans le cas de la non convergence du modèle (voir figure 5.6) c'est à dire qu'il n'est pas possible de borner le jeu de données d'apprentissage à un sous ensemble restreint des traces possibles.

Dans de nombreux cas, il est donc indispensable de généraliser les boucles. La figure 5.7 montre que de cette manière une seule transition est nécessaire pour les représenter. Cette généralisation ne tient cependant pas compte de la réalité d'exécution du programme et donc de ses états pratiques. Dans la réalité, l'exécution de la fonction loop_function ne pourrait prendre qu'un nombre restreint de valeurs comme paramètre, dans ce cas tout autre nombre d'itération de la boucle serait illégitime.

De plus, si le cas d'une boucle classique est relativement simple, il est nécessaire de le différencier d'une répétition de séquence (plusieurs appels linéaires successifs ou boucle de taille fixe) comme celle illustrée dans la figure 5.8. Dans ce cas, il n'est pas

trivial de définir si cette répétition de séquence (non différentiable d'une boucle de taille variable du point de vue des traces) doit être interprétée comme une boucle ou non.

L'algorithme responsable de l'extraction de modèle doit donc trouver le juste milieu entre sur-généralisation et non convergence du modèle en privilégiant la précision tant que cela est possible et en généralisant que lorsque c'est indispensable. Le problème de la convergence a été formalisé par Gold [47] et il est au cœur de la problématique d'apprentissage.

5.1.3 Reconstruction d'un modèle existant

Si l'on revient au cas précis de l'analyse de comportement de programme, il s'agit de créer une modélisation pour un programme. Or comme il a été vu dans le chapitre 2, un programme est une suite d'instructions exécutées par un processeur déterministe et remplissant une fonction définie à l'avance lors de l'écriture du code source. De ce fait, l'exécution du programme suit déjà un modèle existant (le code source), le processus d'apprentissage a donc dans le pire des cas la capacité de converger vers un modèle équivalent au code source. De plus, la fonction du programme étant fixe, celle-ci suit également une logique externe précise, par exemple pour un serveur web, la logique des pages HTML qui sont hébergées dessus suit une logique induite par la présence de liens et de formulaires sur les pages qui s'enchaînent dans un ordre qui n'est souvent pas arbitraire.

De ce fait, il est raisonnable de penser que l'utilisation d'un algorithme d'apprentissage adéquat au problème a de bonnes chances de converger vers un modèle du programme qui serait une synthèse entre la logique induite par le code source et la logique d'utilisation du programme. Le modèle ayant de bonne propriété de convergence nous pensons donc qu'il est possible d'automatiser la création d'un tel modèle via l'utilisation de techniques d'apprentissage automatique.

5.2 Implémentation

5.2.1 A propos de l'apprentissage automatique

Wagner & Dean [102] se sont intéressés à la sécurité des programmes et à la modélisation adéquat pour détecter une déviation dans le comportement attendu. En s'intéressant à la façon dont un programme effectue des appels systèmes ils ont proposé plusieurs modélisation dont l'automate et la grammaire non contextuelle. Ces deux représentations sont en fait équivalentes d'un point de vue sémantique et il existe donc une relation bijective entre les deux : on peut obtenir la grammaire correspondant à un automate et l'automate correspondant à une grammaire. Chercher un algorithme d'apprentissage capable de construire un automate à partir d'une trace revient donc à chercher un algorithme d'apprentissage capable de trouver une grammaire à partir d'un texte.

Ce problème est connu sous le nom d'induction de grammaire ou encore d'inférence grammaticale. Cette problématique est récurrente et donc par conséquent bien documentée [32, 34, 14] . On la retrouve bien sûr lorsque l'on s'intéresse à la modélisation du langage naturel [69, 57, 5] pour comprendre une requête humaine ou traduire un

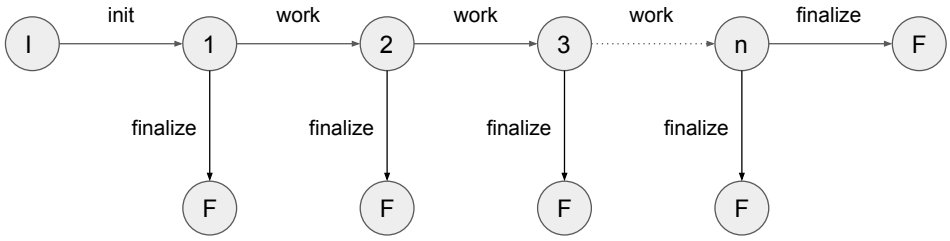

FIGURE 5.6 – Exemple d'automate absolu créant une branche pour chaque valeur possible de la borne de boucle

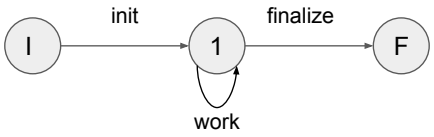

FIGURE 5.7 – Exemple d'automate généralisé

```
void read_files(){
        read(file1);
        read(file2);
        read(file3);
    return;
}
```

FIGURE 5.8 – Exemple de répétition d'appels

texte, mais on peut la retrouver également dans des domaines moins évident comme l'inférence de données statistiques [15], la modélisation de séquence d'ADN [23] ou encore la génération automatique de DTD à partir de fichiers XML [38].

5.2.2 Choix de l'algorithme d'apprentissage

Algorithmes gloutons

Afin de résoudre le problème de l'inférence grammaticale de nombreuses méthodes ont été proposées. Pour commencer, l'utilisation d'algorithmes gloutons [111, 106, 107], utilisés notamment pour la compression, ces algorithmes vont consommer la trace petit à petit et prendre la meilleure décision de généralisation pour le sous ensemble évalué. Comme ils raisonnent sur un sous-ensemble de la trace sans se soucier de son intégralité, les algorithmes gloutons offrent de bonnes propriété de rapidité d'exécution mais sans offrir la garantie d'une solution optimale.

Après quelques tests nous avons rapidement éliminé les algorithmes gloutons. Si cette solution offre l'avantage de pouvoir être exécutée de façon locale (et donc ne nécessite pas de charger l'intégralité de la trace en mémoire) elle est cependant peu précise

Algorithmes RPNI

Un autre solution proposée est l'utilisation d'exemple positifs et négatifs [80, 41], cette solution est appelée RPNI. Cette méthode se base sur la distinction d'exemples positifs (dans notre cas les traces valides) et d'exemples négatifs (les traces invalides). Elle consiste à construire un modèle naïf reconnaissant un premier exemple positif et ensuite faire évoluer ce modèle afin qu'il reconnaisse tous les exemples positifs et rejette tous les exemple négatifs. Dans ce cas, chaque version du modèle peut être vue comme une hypothèse et chaque nouvelle exemple comme un test sur cette hypothèse : si la réponse est correcte l'hypothèse tient toujours sinon l'hypothèse doit être modifiée. Dans le cas de la généralisation de modèle, les algorithmes RPNI vont chercher à généraliser le modèle pour accepter les exemples positifs tout en rejetant les exemples négatifs. Ainsi s'il manque d'exemples positifs l'algorithme ne généralisera pas suffisamment et donc ne convergera pas, tandis qu'en l'absence d'exemples négatifs, l'algorithme aura tendance à sur-généraliser le modèle.

Cette méthode est donc efficace si l'on est capable de fournir un jeu de données représentatif d'exemples positifs et négatifs (à noter qu'il serait nécessaire de définir la nature d'un tel jeu de données représentatif). Cependant, nous voulions une solution entièrement automatisée, et il n'est pas possible de disposer d'un algorithme listant de façon exhaustive l'ensemble des traces invalides. En effet, si nous disposions d'un tel algorithme alors la condition « if(trace $\in$ fausse_traces) » serait un moyen fiable de discriminer automatiquement un comportement invalide d'un comportement valide. Or nous cherchons justement à produire un tel mécanisme automatique de discrimination, parce que nous n'en disposons pas... Nous ne pouvons donc pas postuler que nous trouverons un moyen de produire des exemples de fausses traces automatiquement, pour alimenter notre algorithme d'apprentissage RPNI.

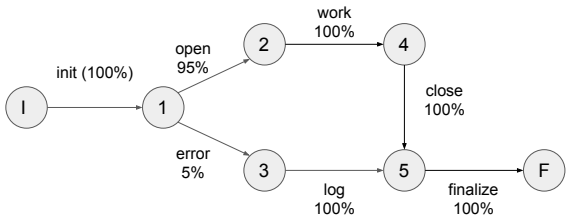

FIGURE 5.9 – Exemple d'automate probabiliste

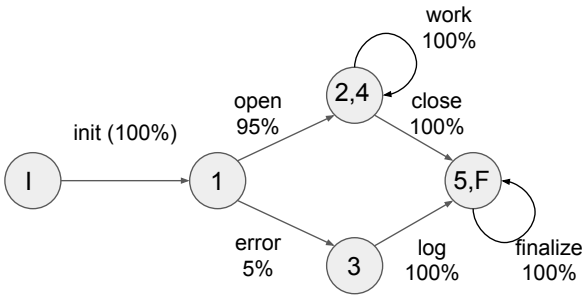

FIGURE 5.10 – Fusion d'état dans un automate probabislite

Algorithmes génétiques et réseaux de neurones

Les algorithmes génétiques [53, 33, 58] et les réseaux de neurones [44, 104] ont été utilisés également pour résoudre le problème de l'inférence grammaticale. Cependant nous voulions commencer par un algorithme plus simple et plus transparent et nous avons donc fait le choix de ne pas les considérer dans le cadre de ces travaux.

Alergia

L'algorithme Alergia [19] est une référence dans l'apprentissage d'automates à partir d'exemples uniquement positifs. Son implémentation repose sur la création et la simplification d'un automate probabiliste fini. L'algorithme attribue à chaque transition dans l'automate une probabilité que celle-ci se produise comme illustré dans la figure 5.9

L'algorithme regroupe ensuite les états en fonction des probabilités de leurs transitions. Deux états successifs ayant des probabilités de transition proches seront fusionnés. Par exemple, dans la figure 5.10, les états 2 et 4 ainsi que les états 5 et F ont étés fusionnés. De plus, l'algorithme permet de spécifier un seuil de probabilité de transition entre deux états à partir duquel il doit fusionner ces états, il est donc possible de ne fusionner que les transitions fortement probables et de garder une précision plus importante pour les transitions plus rares.

Ce modèle d'apprentissage est particulièrement adapté à l'apprentissage de comportement de programme car ces comportements peuvent souvent etres divisés en deux

catégories : les comportements normaux (ceux-ci vont apparaître dans la majorité des traces) et les comportement exceptionnels (comportement d'erreur, mode debug, gestion d'exceptions, taches exceptionnelles, etc...). Dans les comportements normaux, on va retrouver des taches simples, souvent des boucles d'attente ou les traitements courants, alors que dans les comportements exceptionnels on peut s'attendre à trouver des actions plus critiques (envoi de logs, réinitialisations, arrêt d'une fonctionnalité, etc...) qui peuvent perturber fortement le comportement du programme si elles sont exécutées sans raison valable. Ainsi il est donc très intéressant de pouvoir conserver une précision forte sur ces éléments rares tout en généralisant le code courant. Nous avons donc choisi l'algorithme Alergia pour créer notre modèle de référence.

5.3 Évaluation

Afin d'évaluer la pertinence de la démarche d'apprentissage ainsi que du choix d'Alergia comme algorithme d'apprentissage, nous avons évalué les aspects suivants : convergence de l'algorithme d'apprentissage et précision du modèle appris. Ainsi nous pourrons vérifier la pertinence de la démarche d'apprentissage et comparer les résultats obtenus par rapport à une solution utilisée dans l'état de l'art, l'objectif étant que le nouveau modèle réussisse à détecter des chemins invalides là où les modèles traditionnels auraient échoué.

5.3.1 Utilisation du serveur web nginx et fuzzing

Afin de réaliser cette évaluation j'ai choisi de me placer dans un cas concret d'application à surveiller. Mon choix s'est porté sur le serveur web nginx [85], un logiciel open-source portable sur les systèmes embarqués proposant une taille suffisante pour que celui-ci soit non trivial à analyser. De plus nginx possède une chaine de compilation claire et donc facile à instrumenter. Finalement, leur exposition vers l'extérieur en font des cibles privilégiées pour des attaques et l'on trouve donc en réponse un grand nombre de programmes (fuzzer par exemple) permettant de les tester facilement afin de s'assurer de leur fiabilité.

Lors de l'évaluation, les cas d'usage du serveur web ont été limités aux parcours d'un site web via url afin de conserver un cas d'usage facile à comprendre et à appréhender. Pour construire le jeu de données d'apprentissage, j'ai donc utilisé le fuzzer w3af [86] sur un site web de test (une copie locale de www.aidedd.org). Le fuzzer w3af a été utlisé pour sa fonction de *crawler* c'est à dire qu'il parcourt toutes les entrées possibles du site, page par page et recherche également les ressources non référencées. L'utilisation de ce fuzzer permet de représenter une utilisation typique d'un appareil embarqué à savoir d'héberger un site web unique et statique servant de panneau de commande.

5.3.2 Évaluation de la convergence

Comme expliqué dans la section 5.1.2, tout le défi de la phase d'apprentissage était de parvenir à la création d'un modèle le plus précis possible tout en conservant une bonne propriété de convergence.

Afin d'évaluer la convergence nous avons observé la capacité du modèle à reconnaître l'ensemble des traces valides. Le protocole expérimental est le suivant : une portion de

plus en plus grande du total de trace est présentée à l'algorithme d'apprentissage afin de construire un modèle de ces traces à l'aide de l'algorithme Alergia. Le modèle créé par Alergia est ensuite chargé dans le moniteur. Finalement, on présente au moniteur l'ensemble des traces et celui-ci vérifie si elles sont valides vis à vis du modèle. Cette démarche permet donc d'avoir une estimation de la proportion de trace nécessaire à l'algorithme d'apprentissage afin de créer un modèle capable de reconnaître un ensemble de traces légitimes.

Quelques chiffres

Les tests suivants ont étés réalisés à partir d'un *fuzzing* de type *crawler* sur une version locale du site aidedd.org hébergé sur un serveur nginx. La version du site utilisée pèse 150mo et comprend 3300 fichiers dont 1314 pages html. Lors d'un fuzzing d'une durée de 5 minutes, 22 traces d'exécutions représentant un total de 540mo de traces ont été générées ce qui représente douze millions d'appels de fonction répartis sur 491 fonctions. Enfin parmi ces appels on a dénombré 764 exécutions uniques de fonction.

Résultats

Les résultats attendus sont une diminution rapide du nombre d'erreur ce qui signifierait que le modèle tend rapidement vers un modèle complet et stable.

Le graphique 5.11 montre l'évolution du nombre d'erreur en fonction de l'augmentation de la taille de l'échantillon d'apprentissage. Comme on peut le voir, le taux de précision initial est de 95,9% ce qui veut dire qu'en analysant uniquement une seule trace de chaque fonction, le modèle est capable de reconnaître 95,9% des traces qui lui seront présentées. Ce taux monte à 99% si l'on utilise 10% du total de traces pour l'apprentissage. Ce résultat est en partie dû au fait que de nombreuses fonctions ont des comportements identiques à chaque exécution.

Sur les 486 fonctions du serveur nginx qui ont été monitorées on dénombre :
- 166 fonctions linéaires, celles-ci produisent toujours la même trace
- 224 fonctions ayant uniquement 2 exécutions possibles
- 75 fonctions ayant entre 3 et 4 exécutions possibles
- 21 fonctions ayant 6 exécutions possibles ou plus

Si l'on répète l'expérience en considérant uniquement les fonctions ayant au moins 5 exécutions possibles alors la progression de l'apprentissage est plus lente comme on peut le voir sur le graphique 5.12. Cependant, celle-ci suit la même courbe asymptotique et avec 10% des traces utilisées on obtient 90% de précision.

L'ensemble des résultats en courbe asymptotique montre cependant une nette convergence vers un modèle stable.

5.3.3 Comparaison avec l'existant

Afin de comparer notre approche avec les modèles existant nous avons extrait les graphes de flot de contrôle correspondant aux fonctions représentées par nos automates. Dans cette comparaison nous avons évalué 417 automates et les 417 CFG correspondants.

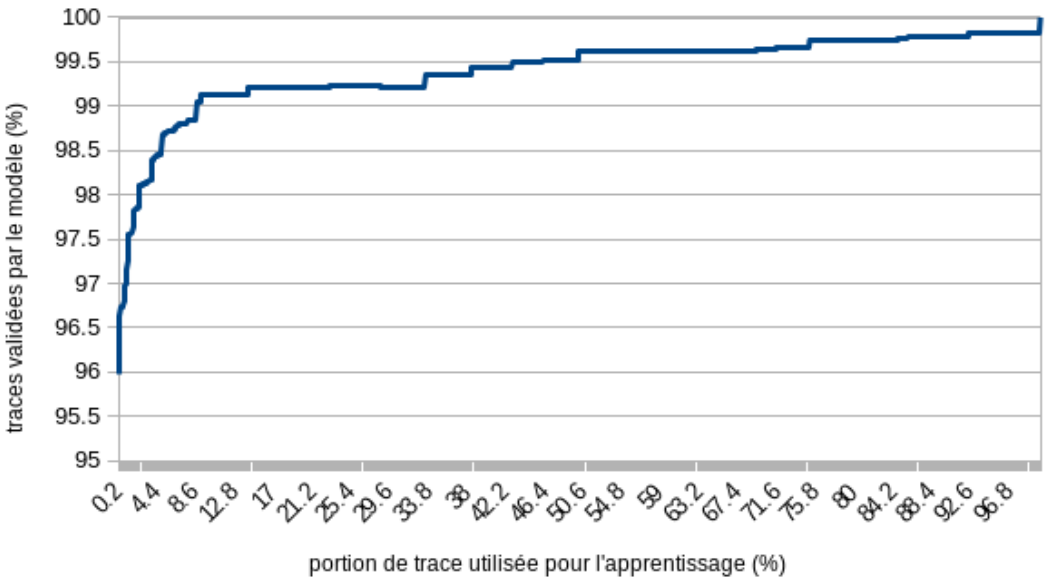

FIGURE 5.11 – Précision du modèle en fonction de la proportion de trace utilisée pour l'apprentissage

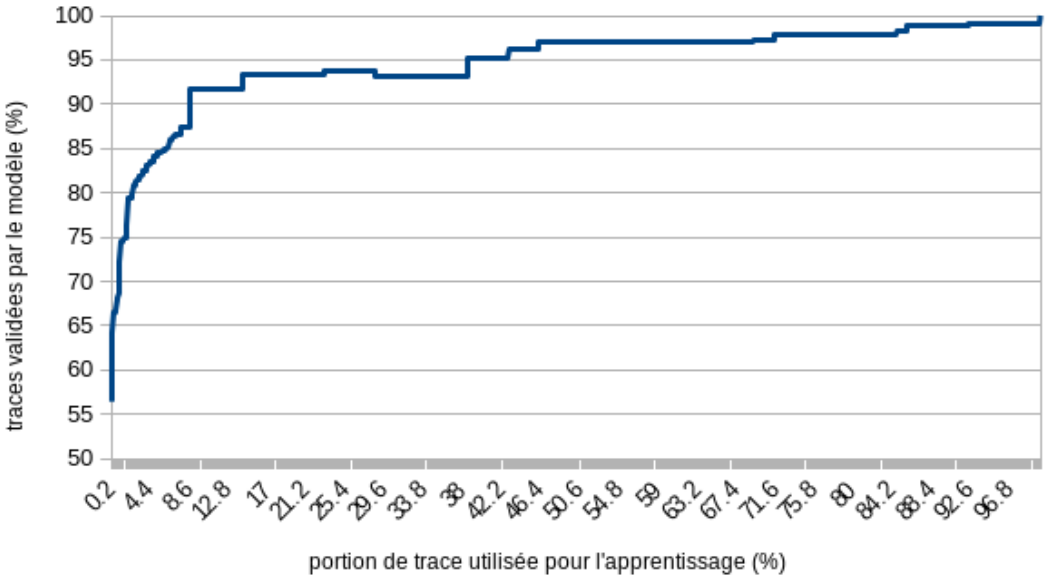

FIGURE 5.12 – Précision du modèle en fonction de la proportion de trace utilisée pour l'apprentissage pour l'apprentissage de fonctions complexes

```
void function(){

        print("this");
        print("is");
        print("a test");
        do_work();

        return;
}
```

FIGURE 5.13 – Fonction réalisant deux appels distincts : print et do_work

Après avoir analysé l'ensemble des automates produits, il en ressort que ceux-ci référencent 374 appels distincts de fonction cumulés sur les 417 automates produits. Nous définissons un appel distinct comme étant la première fois qu'une nouvelle fonction est référencée dans un automate (c'est à dire la première fois qu'une fonction déclenche une transition vers un nouvel état). La figure 5.13 montre un exemple de fonction réalisant deux appels distincts. L'utilisation de cette mesure a pour but de ne pas être impactée par les simplifications faites lors de multiples appels à une même fonction.

Si l'on compare ce chiffre avec celui qui ressort de l'analyse des CFG on trouve 795 appels distincts cumulés parmi les 417 CFG. Cette différence s'explique par le fait que le CFG représente un sur-ensemble des fonctionnalités testées et donc incluant du code non utilisé. Notre cas d'étude était un fuzzing de type crawler sur un serveur web, ce cas d'étude ne couvre donc pas toutes les possibilités offertes par l'utilisation exhaustive d'un serveur web. Ce chiffre permet donc de mettre en évidence l'étendue du code inutile (421 appels distincts) qui peut être embarqué sur un dispositif, ce code n'étant d'aucune utilité pour le bon fonctionnement de l'appareil mais restant à la disposition de l'attaquant pour une attaque par réutilisation de code.

En complément des 374 appels distinct communs, les automates ont également détecté 428 appels par pointeur de fonctions, non détectés par les CFG. Lors de l'exécution en conditions saines, il y a donc un total de 802 appels distincts qui ont étés détectés dont plus de la moitié issus de pointeurs de fonctions. Ce chiffre nous permet de constater l'apport considérable que fournit l'apprentissage automatique dans la prédiction de cibles de branchements indirects et l'impact que peut avoir une imprécision sur ces pointeurs.

Pour conclure sur la précision, l'évaluation a permis de mettre en évidence l'étendue du code inutilisé qui était ajouté aux modèles créés par analyse de code ainsi que le nombre important d'appels indirects qui ne sont pas identifiés précisément dans ces modèles. La démarche d'apprentissage semble donc vraiment prometteuse afin de résoudre les problématiques de précision dont souffrent les modèles de références dans les CFI actuels.

5.3.4 A propos des résultats

Lors de cette évaluation, nous avons vu que le choix de l'algorithme d'apprentissage automatique Alergia permettait de produire un automate déterministe fini. L'apprentissage de cet automate converge vers un modèle stable selon une courbe d'apprentissage asymptotique. L'automate produit en sortie offre une précision au moins équivalente

à celle d'un graphe de flot de contrôle tout en intégrant les branchements indirects et en éliminant le code non utilisé. Cette nouvelle catégorie de modèles peut donc être utilisée dans les CFI et ainsi offrir une meilleur précision de surveillance.

5.4 Perspectives

Dans ce chapitre il a été démontré qu'il est possible de créer, à partir de traces réelles d'exécution, un modèle servant de référence à une solution de type CFI. L'utilisation d'un algorithme d'induction de grammaire basé sur les automates probabilistes permet de créer un modèle simplifié et donc convergent tout en conservant la logique du programme qu'il représente et les spécificités de celui-ci. De plus, le modèle construit offre une meilleure précision qu'un CFG classique tout en réglant de façon conceptuel les problèmes de prédiction des branchements indirects. Enfin les tests en condition réelle montrent une bonne convergence du mécanisme d'apprentissage ainsi qu'une couverture de sécurité reprenant toutes les forces du CFI en gommant un grand nombre de ses faiblesses.

Il reste néanmoins beaucoup de pistes à explorer. Premièrement, le choix de l'algorithme d'apprentissage peut être affiné en testant différents algorithmes et en comparant les résultats. En allant plus loin il serait intéressant d'examiner les algorithmes d'apprentissage fonctionnant sur des automates d'arbres. En effet ceux ci permettraient l'ajout d'une nouvelle dimension, on retrouverait toujours l'ordre des appels au sein d'une fonction dans l'horizontalité de l'arbre mais également la notion de contexte d'appel dans sa verticalité. Ceci permettrait de différencier deux appels et donc deux traces suivant la fonction qui les a appelé, ce qui apporterait une finesse de surveillance tout à fait nouvelle.

Afin de compléter ces recherches, il faudrait réaliser une étude sur les *fuzzer* et les autres méthodes de génération de traces de test. Cette étude aurait pour but de connaître plus précisément le lien entre la génération de traces et l'apprentissage afin de déterminer les bonnes et mauvaises pratiques pour optimiser la phase d'apprentissage et obtenir le meilleur rapport entre précision et convergence possible.

Chapitre 6

Conclusion

6.1 Résumé des contributions

Dans cette thèse, je me suis intéressé à la sécurité des systèmes embarqués. J'ai proposé EE-CFI [62], une solution externalisée de CFI basée sur la production d'une trace d'exécution et sur l'analyse de cette trace depuis l'extérieur par un moniteur de surveillance. J'ai évalué la faisabilité de la démarche en démontrant que si l'on restreint la surface à surveiller à un sous ensemble du code, alors la solution est adaptée aux contraintes de performance des systèmes embarquées. J'ai également proposé une amélioration du modèle de référence [63] utilisé par le moniteur dans le but de palier aux faiblesses des solutions actuelles de CFI. Cette nouvelle approche pour la création du modèle de référence est basée sur l'apprentissage automatique de comportement. Elle repose sur l'utilisation de l'algorithme Alergia afin d'extraire un modèle à partir de traces d'exécutions saines. J'ai évalué la pertinence de cette approche en évaluant la capacité d'Alergia à construire rapidement un modèle cohérent capable de valider l'ensemble des traces d'exécution légitimes. J'ai ensuite comparé le nouveau modèle à un modèle classique de CFG et celui-ci montre des résultats très prometteurs en affichant une précision supérieure au CFG tout en répondant au problème de prédiction des branchements indirects dans le CFI.

6.2 Retour sur la problématique

Dans le chapitre 2, j'ai dressé le bilan de la sécurité des systèmes embarqués à travers le prisme de l'intégrité du flot de contrôle. Nous avons vu tout d'abord que la bataille de l'intégrité du flot de contrôle avait pris des allures de guerre de tranchées ces dix dernières années. D'un coté de nombreuses solutions de CFI aux méthodes variées ne parvenant pas à faire l'unanimité en raison d'un besoin de compromis entre sécurité et performance. De l'autre coté, on observe des attaques par réutilisation de code toujours plus élaborées mais également de plus en plus difficiles à mettre en place. Ces attaques misent d'une part sur une utilisation frauduleuse de transitions paraissant légitimes et d'autre part sur l'imprédictibilité de certains comportements du programme, en particulier les branchements indirects (i.e. les pointeurs de fonction). Et c'est entre ces deux camps que se trouvent les systèmes embarqués, qui pas assez puissants pour intégrer de coûteuses solutions de CFI et qui sont donc complètement à découvert contre le feu ennemi suffisamment équipé pour défaire facilement leurs maigres défenses. Derrière cette métaphore se trouve un enjeu bien réel, celui de sécuriser des systèmes toujours plus présents dans notre quotidien et donc toujours plus proches de nous. De plus de

par leurs conditions de développement, ces systèmes ne permettent souvent pas une mise à jour des vulnérabilités et ils sont donc d'autant plus sensibles aux attaques. Nous avons dressé le constat que, pour proposer une solution qui se voudrait universelle pour ces systèmes hétéroclites, reposant sur un matériel varié et n'ayant peu ou pas de surcouche logicielle, il faut proposer une solution intervenant dans le processus de compilation du code embarqué. Nous avons vu également qu'en raison des maigres mécanismes de protection matériel à leur disposition, il faudrait isoler le mécanisme de protection si l'on veut que celui-ci soit efficace.

En réponse à cela, nous avons proposé une solution externalisée de surveillance du flot de contrôle reprenant les principes du CFI en utilisant un moniteur externe afin de surveiller le comportement de l'appareil. Cette solution, intégrée lors de la compilation, répond à la problématique de performance en limitant au strict minimum la tache de l'appareil embarqué dans le processus de sécurité. En déléguant l'analyse de son comportement à un dispositif externe, le système embarqué peut ainsi bénéficier d'une excellente protection à moindre coût. Cette externalisation permet également de se prémunir contre les attaques qui cibleraient directement le mécanisme de défense, répondant ainsi également à la problématique d'isolation de la solution de sécurité.

Pour finir, nous avons démontré que les systèmes embarqués possèdent leurs propres forces dans la bataille de la sécurité. Grâce à leur surface de code réduite et à leur processus de développement, il est possible d'utiliser un algorithme d'apprentissage afin d'induire le modèle de référence de la solution de CFI à partir de traces d'exécutions saines. Après avoir démontré la cohérence de mécanisme d'apprentissage à travers une évaluation de sa convergence, nous avons montré que ce nouveau modèle est plus précis que les modèles classiques de CFI, réduisant de fait la marge de manœuvre de l'attaquant. Nous avons également montré que cette approche permet de résoudre une grande partie des problèmes d'imprédictibilité du flot de contrôle en identifiant clairement les cibles de branchements indirects.

Notre approche permet donc de proposer une réponse adaptée et réaliste aux problèmes de sécurité du flot de contrôle sur les systèmes embarqués. Celle-ci ouvre également des perspectives pour l'avenir de la sécurité en ouvrant la porte de l'apprentissage automatique orienté sécurité sur le monde de l'embarquée. Ce travail est avant tout un travail préliminaire, son objectif étant de répondre à la question de la faisabilité de ces solutions et c'est donc tout naturellement qu'il se conclut en ouvrant une multitude de nouvelles questions.

6.3 Perspectives

Dans la section précédente, nous avons vu que ce travail répond à deux problématiques principales : adapter la sécurité à l'informatique embarqué et améliorer le modèle de CFI. Pour la suite de ces travaux de nombreuses pistes peuvent être envisagées. Dans cette section nous allons lister ces pistes de manière chronologique en commençant par les pistes à court terme.

6.3.1 A court terme

Ces pistes sont à évaluer en premier lieu car elles sont nécessaires à la mise en production de la solution. Elles relèvent de problématiques qui sont avant tout industrielles.

Evaluation du moniteur : Dans ces travaux, notre objectif était de montrer de faisabilité de l'externalisation. Pour cela, nous avons utilisé un prototype simple de moniteur sous la forme d'un programme python. Afin d'envisager un déploiement concret de la solution, il faudrait évaluer plus en détail le matériel adapté à la solution. De ce coté plusieurs pistes sont envisagées. Premièrement l'utilisation d'un serveur distant, celui-ci aurait l'avantage de pouvoir centraliser la vérification d'un grand nombre de cibles, possiblement dispersées sur plusieurs sites géographiques. Le point faible de cette solution vient de l'utilisation d'internet pour la communication, celle-ci pourrait en effet être facilement interceptée ou corrompue grâce à une attaque de type *man-in-the-middle*. La deuxième option envisagée est l'utilisation d'un serveur centralisé local, utilisant un réseau privé (dédié ou non) qui serait donc inaccessible depuis l'extérieur. Enfin les solutions mono-cibles, l'utilisation d'un second système embarqué (microcontrôleur ou carte à puce) dédié à la surveillance de la cible. L'évaluation et le choix de la solution moniteur est intrinsèquement liée au contexte d'utilisation de la cible, une étude exhaustive de ces solutions devrait prendre en compte le nombre de cibles, leurs spécifications techniques, la criticité des opérations effectuées par les cibles ainsi que la confidentialité des informations qu'elles traitent.

Evaluation de la communication : Cette étude démontre que le volume de données générées par la surveillance est acceptable pour une utilisation par des systèmes embarqués classiques. Cependant, comme nous venons de le voir, en plus de devoir s'adapter à la cible, le canal de communication varie en fonction du moniteur choisi. Suivant le canal de communication choisi, son taux de perte, son accessibilité depuis l'extérieur et les protocoles sur lesquels il repose, la solution de communication devra être adaptée. L'utilisation de protocoles de chiffrement ou de vérification d'intégrité des données reçues pourront s'avérer nécessaires et il faudrait alors en évaluer le coût sur la cible ainsi que sur le moniteur.

Utilisation d'autres algorithmes d'apprentissage et paramétrage : Lors de l'évaluation de l'apprentissage nous avons utilisé l'algorithme Alergia avec un paramétrage adapté au jeu de données que nous lui avons fourni. Dans des travaux futurs, il serait nécessaire d'évaluer un ensemble d'algorithmes d'apprentissage et leur paramétrage dans différents cas d'utilisation afin de voir si il existe une solution optimale adaptée à tous les cas d'usages ou si au contraire il faut adapter la méthode d'apprentissage au code que l'on souhaite surveiller. De plus les méthodes d'apprentissage plus complexes, utilisant des algorithmes génétiques ou des réseaux de neurones n'ont pas été évaluées dans cette étude, il serait donc intéressant de voir les résultats que ces algorithmes peuvent fournir et de les comparer aux résultats fournis par Alergia.

Construction du jeu de données pour l'apprentissage : La création du jeu de données influence de façon directe la faculté d'apprentissage et la cohérence du modèle produit. Pour cette étude, nous avons utilisé un fuzzer web de type crawler et notre jeu de données concernait donc l'ensemble des opérations effectuées par nginx pour répondre aux requêtes sur le site de test. Comme il a été détaillé dans le chapitre 3, d'autres méthodes peuvent être envisagées pour la production du jeu de données comme l'utilisation d'un jeu de tests unitaires et/ou fonctionnels ou l'utilisation d'un fuzzer semi-guidé. Afin de s'assurer d'avoir une phase d'apprentissage optimale, il faudra évaluer l'impact concret de la composition du jeu de données sur les différents algorithmes d'apprentissage

afin de trouver la combinaison optimale algorithme/jeu de données en fonction de la cible que l'on souhaite surveiller.

Ces pistes de recherche représentent donc la base minimale à évaluer afin d'envisager un déploiement concret de la solution proposée dans ce document.

6.3.2 A moyen terme

Ces pistes ne sont pas nécessaires à la mise en place de la solution, néanmoins elles représentent des opportunités concrètes d'amélioration de la solution.

Mode diagnostique du moniteur : Lorsqu'une vulnérabilité est détectée par le moniteur, celui-ci nous informe que le système a été compromis et que l'appareil embarqué n'est donc plus fiable. Cependant il serait intéressant de fournir un rapport plus détaillé de la compromission afin de pouvoir analyser l'attaque et si possible de pouvoir corriger le code. Ce rapport pourrait inclure plusieurs informations comme par exemple l'emplacement de la rupture de flot de contrôle, l'état du système à cet instant mais également un rapport plus détaillé des opérations effectuées après la rupture afin de déterminer exactement le contenu de l'attaque. Cette fonctionnalité pourrait également servir à diagnostiquer des dysfonctionnements de l'appareil suite à des erreurs dans le code. Le moniteur pourrait également intégrer un mécanisme d'ordre à destination de la cible afin de demander à celle-ci de se couper ou de redémarrer. Cette intervention pourrait se faire avec une coupure de l'alimentation ou en utilisant la fonction *wake-on-lan* permettant de redémarrer un appareil via une trame Ethernet spécifique.

Compression de la trace : Comme nous l'avons vu dans le chapitre 5, une grande partie de la trace d'exécution est composée d'appels répétés à des fonctions simples. De plus, les systèmes embarqués ayant souvent pour but d'effectuer des taches simples en continu, la trace est donc extrêmement répétitive. Lors de travaux futurs, il sera intéressant de se pencher sur cette spécificité afin d'évaluer si il est possible de simplifier la trace pour réduire le temps d'apprentissage. La cible pourrait ignorer des séquences répétitives spécifiques afin de fournir un jeu de données d'apprentissage plus compact. De plus, lors de l'exécution, la trace pourrait être également simplifiée afin de n'être envoyée en vérification que si un changement de comportement est détecté. Il faudrait alors évaluer l'impact que cette fonction aurait sur la sécurité en s'assurant que cette fonctionnalité ne peut pas être utilisée à des fins malveillantes par l'attaquant.

Surveillance du contexte d'appel : Dans la section 5.4 nous avons proposé l'utilisation d'automates d'arbres afin d'ajouter une dimension supplémentaire au modèle. Cette nouvelle dimension, représentant concrètement le contexte d'appel d'une fonction, permettrait de réduire considérablement la marge de manœuvre de l'attaquant. Leur utilisation offrirait une amélioration certaine du modèle car elle offrirait alors une précision tout à fait nouvelle dans la finesse de surveillance sans augmenter le coût pour la cible. Afin d'évaluer la pertinence de cette approche il faudrait évaluer la variance de la trace et voir si celle-ci est liée au contexte d'appel. Il faudrait également pour cela adapter les algorithmes d'apprentissage et évaluer de nouveau leur convergence.

6.3.3 Autres pistes de réflexion

Pour terminer cette section, deux autres pistes plus vagues ont été envisagées durant cette étude. Ces pistes sont moins concrètes et leur réflexions moins abouties que celles citées précédemment.

Apprentissage de données : Lors de ces travaux, nous avons considéré uniquement le flot de contrôle du programme comme représentation du programme à surveiller. D'autres aspects peuvent être pris en compte et notamment la surveillance des données du programme. A l'image des pointeurs de fonction, l'évolution du contexte du programme est délicate à prédire statiquement. Or, comme nous l'avons vu pour les pointeurs de fonction, l'utilisation de l'apprentissage permet de prédire l'évolution de ces variables de façon concrète. Il est donc envisageable d'élargir cette approche à d'autres données du programme telles que des valeurs de compteur sensées suivre une incrémentation stricte, des valeurs ayant des bornes précises ou même la corrélation de plusieurs valeurs.

Apprentissage de comportement commun : Dans ce chapitre nous avons proposé la mutualisation de la protection de plusieurs systèmes embarqués par un moniteur commun. Cette approche peut être poussée en mutualisant le comportement des différents appareils afin d'extraire un modèle commun d'exécution. En effet, dans le cas d'un réseau de capteurs, leurs exécutions peuvent probablement être corrélées et il serait donc possible d'identifier la dérive d'un des capteurs en comparant son comportement à celui des capteurs environnants. De plus cette approche pourrait, dans les cas de systèmes critiques, reposer sur l'utilisation d'un duo de systèmes identiques, effectuant la même tache et dont les comportements seraient comparés l'un à l'autre.

6.4 Retour d'expérience personnelle

Afin de contextualiser ce retour d'expérience, je dois préciser que lorsque j'ai commencé mes travaux sur la sécurité des systèmes embarqués je n'avais que très peu de connaissance sur la sécurité informatique et sur les systèmes embarqués. Ce travail à donc été pour moi, ayant une formation en génie logiciel, une découverte totale de ces domaines et de la relation si particulière qui les lie. C'est donc avec un œil neuf, parfois même un peu naïf, mais dénué d'idées préconçues, que j'ai abordé ces recherches. Au cours de cette thèse, j'ai du faire un certain nombre de choix afin de définir les grands axes de mon travail. Ces choix ont en partie été motivés par l'état de l'art sur le sujet ainsi que par les résultats obtenus mais également en fonction des compétences à ma disposition, aussi bien personnelles que dans mon entourage scientifique.

Le premier choix que j'ai du faire était dans l'interprétation même du sujet à ma disposition. Nous nous intéressions à l'époque à la recherche par analyse statique de code d'invariants dans les noyaux de systèmes d'exploitation. Vaste sujet, je n'avais qu'une connaissance légère et essentiellement théorique des noyaux de systèmes et j'ai donc fait le choix de conserver cette approche théorique, sans rentrer dans les détails d'implémentation d'un système spécifique. Je me suis alors intéressé au fonctionnement général d'un système d'exploitation, à la recherche d'invariants. Après avoir passé en revue les principales attaques et défenses des noyaux de systèmes, j'ai vite remarqué que beaucoup s'étaient intéressés au flot de contrôle comme garant de la bonne exécution d'un système. Cette approche m'a intéressé par son aspect universel,

en effet l'analyse comportementale de programme trouve de nombreuses applications dans la recherche de bogue, l'analyse de malware, l'optimisation de performances entre autres. J'avais déjà, au cours de ma formation, eu l'opportunité de m'intéresser aux comportement d'un programme en réalisant une implémentation des travaux de Cohen [24]. Cette approche m'était donc familière et j'avais l'intuition que c'était prometteur, personne n'avait encore proposé de vérification externalisée du flot de contrôle.

Seulement après une recherche plus approfondie sur le sujet et sur les noyaux de systèmes d'exploitation, je me suis vite rendu compte que le sujet ne serait pas aussi trivial qu'il n'y parait et que j'aurais à adresser un nombre très important de problèmes dus aux spécificités même de ces systèmes surtout si l'on s'intéresse aux noyaux monolithiques des systèmes massivement déployés que sont Linux et Windows. Il fallait que je commence par quelque chose de plus petit, de plus abordable. Je me suis donc intéressé aux systèmes embarqués. J'avais eu un aperçu de leur fonctionnement durant ma formation en utilisant un mbed [68] équipé du serveur web smews [35]. Après avoir passé en revu l'état de la sécurité de ces systèmes il m'est apparu évident que c'était la bonne démarche à suivre. Ces systèmes fragiles produits à des centaines de milliers d'exemplaires avec du code impossible à mettre à jour étaient dispersés dans la nature et les premières études pessimistes commençaient à entrevoir ce qui donnerait vie quelques temps plus tard au Botnet Mirai et à une multitude d'autres attaques. Le grille pain démoniaque (*The Evil Toaster*) des scénarios catastrophe s'était transformé en poupée connectée espionnant les enfants à l'insu de leur parents et avec elle une multitude de nouvelles attaques qui semblent tout droit sorties d'un scénario de la série *Black Mirror*. Pas de doute, la problématique de la sécurité de l'IoT doit devenir une préoccupation majeur dans le domaine de la sécurité et dès lors je ne me voyais plus travailler sur un autre sujet que celui ci.

A ce stade, j'avais obtenu suffisamment de retour sur la sécurité pour comprendre que deux approches s'offraient à moi. La première approche était celle de la sécurité pure et dure, la sécurité mathématique, celle que l'on peut prouver. Qui n'a jamais rêvé, en faisant de la sécurité, de pouvoir affirmer : "c'est bon, j'ai résolu le problème, plus aucun attaquant ne pourra jamais passer par là". Si cette affirmation semble indispensable à la conception d'une solution de sécurité, la réalité est en revanche beaucoup plus contrastée. Il est en effet extrêmement difficile de prouver une affirmation dans le domaine de la sécurité et parmi les propositions sur le sujet qui ont été faites, celles-ci reposent toutes sur une série de prédicats stricts et tout compte fait peu réalistes à savoir une intégrité absolue du matériel, l'absence de fautes, de pannes ou de toute autre interférence qui viendrait mettre à mal les rouages de la preuve. Cette approche n'en reste pas moins pertinente, car si en effet on ne peut pas tout prouver, la preuve de certaines propriétés dans un système peut offrir des garanties fortes de sécurité. La deuxième approche est celle de la sécurité empirique, expérimentale. Cette fois-ci, on abandonne toute prétention de prouver quoi que ce soit et l'on se contente de montrer que, sous certaines conditions expérimentales, ça semble fonctionner. Cette approche est beaucoup plus proche de l'approche utilisée pour monter une attaque. La sécurité devient alors un jeu du chat et de la souris, proposant des défenses en constantes évolution contre une attaque qui en fait tout autant. Pour mes travaux j'ai fait le choix de l'approche expérimentale.

Après avoir démontré qu'il était possible d'externaliser la vérification du flot de contrôle d'un système embarqué et après avoir dressé un bilan peu encourageant

de l'efficacité des CFI pour sécuriser le flot de contrôle, j'ai réfléchi aux possibilités d'amélioration que l'on pourrait proposer. Toujours en puisant dans mon bagage universitaire, j'ai choisi d'utiliser l'apprentissage automatique afin de modéliser le comportement du programme. Ces techniques étaient effet utilisées dans le domaine du génie logiciel afin d'effectuer de la recherche et de la correction automatique de bogues et de vulnérabilités. De plus, les techniques d'apprentissage devenant de plus en plus performantes (en 2016 le programme AlphaGo battait le champion du monde de Go Lee Sedol), j'avais le sentiment que la guerre de la sécurité finirait tôt ou tard par se jouer sur le terrain de l'intelligence artificielle, autant prendre de l'avance.

Pour résumer, cette thèse m'a permis de monter en compétences dans le domaine de la sécurité informatique et d'utiliser mes connaissances en génie logiciel afin de proposer une réponse innovante à des problématiques d'actualité et d'avenir. Je suis plutôt satisfait de mon travail et des résultats qu'il a fourni et j'espère qu'il sera suivi de bien d'autres travaux sur le sujet.

Bibliographie

[1] URL : https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.8 (visité le 13/10/2018).

[2] *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. URL : https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in (visité le 13/10/2018).

[3] Martín Abadi et al. « Control-flow Integrity ». In : *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. New York, NY, USA : ACM, 2005, p. 340–353. ISBN : 978-1-59593-226-6. DOI : 10.1145/1102120.1102165. URL : http://doi.acm.org/10.1145/1102120.1102165 (visité le 18/02/2016).

[4] Martín Abadi et al. « Control-flow Integrity Principles, Implementations, and Applications ». In : *ACM Trans. Inf. Syst. Secur.* 13.1 (nov. 2009), 4 :1–4 :40. ISSN : 1094-9224. DOI : 10.1145/1609956.1609960. URL : http://doi.acm.org/10.1145/1609956.1609960 (visité le 13/10/2018).

[5] Pieter W Adriaans, Menno Van Zaanen et al. « Computational Grammar Induction for Linguists. » In : *Grammars* 7 (2004), p. 57–68.

[6] Periklis Akritidis et al. « Baggy Bounds Checking : An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. » In : *USENIX Security Symposium*. 2009, p. 51–66. URL : https://www.usenix.org/legacy/event/sec09/tech/full_papers/sec09_memory.pdf (visité le 18/02/2016).

[7] P. Akritidis et al. « Preventing Memory Error Exploits with WIT ». In : *IEEE Symposium on Security and Privacy, 2008. SP 2008*. 2008, p. 263–277. DOI : 10.1109/SP.2008.30.

[8] Manos Antonakakis et al. « Understanding the mirai botnet ». In : *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, p. 1093–1110.

[9] Piotr Bania. « Security mitigations for return-oriented programming attacks ». In : *arXiv preprint arXiv :1008.4099* (2010). URL : http://arxiv.org/abs/1008.4099 (visité le 13/06/2016).

[10] A. Bittau et al. « Hacking Blind ». In : *2014 IEEE Symposium on Security and Privacy (SP)*. 2014, p. 227–242. DOI : 10.1109/SP.2014.22.

[11] *Black Hat USA 2015 : The full story of how that Jeep was hacked*. https://www.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/9493/.

[12] Tyler Bletsch et al. « 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. New York, NY, USA : ACM, 2011, p. 30–40. ISBN : 978-1-4503-0564-8. DOI : 10.1145/1966913.1966919. URL : http://doi.acm.org/10.1145/1966913.1966919 (visité le 18/02/2016).

[13] E. Bosman et H. Bos. « Framing Signals - A Return to Portable Shellcode ». In : *2014 IEEE Symposium on Security and Privacy (SP)*. 2014, p. 243–258. DOI : 10.1109/SP.2014.23.

[14] Eric Brill. « Automatic Grammar Induction and Parsing Free Text : A Transformation-based Approach ». In : *Proceedings of the 31st Annual Meeting on Association for Computational Linguistics*. ACL '93. Stroudsburg, PA, USA : Association for Computational Linguistics, 1993, p. 259–265. doi : 10.3115/981574.981609. url : https://doi.org/10.3115/981574.981609 (visité le 09/03/2018).

[15] Ralf D Brown. « Transfer-rule induction for example-based translation ». In : *Proceedings of the MT Summit VIII Workshop on Example-Based Machine Translation*. 2001, p. 1–11.

[16] Erik Buchanan et al. « When good instructions go bad : Generalizing return-oriented programming to RISC ». In : *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, p. 27–38. url : http://dl.acm.org/citation.cfm?id=1455776 (visité le 13/07/2016).

[17] Nicholas Carlini et David Wagner. « ROP is Still Dangerous : Breaking Modern Defenses ». In : 2014, p. 385–399. isbn : 978-1-931971-15-7. url : https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini (visité le 18/02/2016).

[18] Nicholas Carlini et al. « Control-Flow Bending : On the Effectiveness of Control-Flow Integrity ». In : 2015, p. 161–176. isbn : 978-1-931971-23-2. url : https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini (visité le 18/02/2016).

[19] Rafael C Carrasco et José Oncina. « Learning stochastic regular grammars by means of a state merging method ». In : *International Colloquium on Grammatical Inference*. Springer. 1994, p. 139–152.

[20] Stephen Checkoway et al. « Return-oriented Programming Without Returns ». In : *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. New York, NY, USA : ACM, 2010, p. 559–572. isbn : 978-1-4503-0245-6. doi : 10.1145/1866307.1866370. url : http://doi.acm.org/10.1145/1866307.1866370 (visité le 13/10/2018).

[21] Gang Chen et al. « SafeStack : Automatically Patching Stack-Based Buffer Overflow Vulnerabilities ». In : *IEEE Transactions on Dependable and Secure Computing* 10.6 (nov. 2013), p. 368–379. issn : 1545-5971. doi : 10.1109/TDSC.2013.25.

[22] Yueqiang Cheng et al. « ROPecker : A generic and practical approach for defending against ROP attack ». In : (2014). url : http://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=2972&context=sis_research (visité le 18/02/2016).

[23] Neva Cherniavsky et Richard Ladner. « Grammar-based compression of DNA sequences ». In : *DIMACS Working Group on The Burrows-Wheeler Transform* 21 (2004).

[24] Frederick B Cohen. « Operating system protection through program evolution. » In : *Computers & Security* 12.6 (1993), p. 565–584.

[25] Mauro Conti et al. « Losing Control : On the Effectiveness of Control-Flow Integrity Under Stack Attacks ». In : *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA : ACM, 2015, p. 952–963. isbn : 978-1-4503-3832-5. doi : 10.1145/2810103.2813671. url : http://doi.acm.org/10.1145/2810103.2813671 (visité le 18/02/2016).

[26] Victor Costan et Srinivas Devadas. « Intel SGX Explained. » In : *IACR Cryptology ePrint Archive* 2016.086 (2016), p. 1–118.

[27] Thomas Coudray, Arnaud Fontaine et Pierre Chifflier. « Picon : Control Flow Integrity on LLVM IR ». In : (). url : https://www.sstic.org/media/SSTIC2015/SSTIC-actes/control_flow_integrity_on_llvm_ir/SSTIC2015-

Article-control_flow_integrity_on_llvm_ir-fontaine_chifflier_coudray_esfrDAl.pdf (visité le 18/02/2016).

[28] Crispan Cowan et al. « StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. » In : *Usenix Security*. T. 98. 1998, p. 63–78. URL : https://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf (visité le 18/02/2016).

[29] Joan Daemen et Vincent Rijmen. *The design of Rijndael : AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[30] Lucas Davi, Ahmad-Reza Sadeghi et Marcel Winandy. « ROPdefender : A detection tool to defend against return-oriented programming attacks ». In : *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 2011, p. 40–51.

[31] Lucas Davi et al. « Stitching the Gadgets : On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection ». In : 2014, p. 401–416. ISBN : 978-1-931971-15-7. URL : https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi (visité le 18/02/2016).

[32] Colin De La Higuera. « Current Trends in Grammatical Inference ». en. In : *Advances in Pattern Recognition*. Sous la dir. de Gerhard Goos et al. T. 1876. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 28–31. ISBN : 978-3-540-67946-2 978-3-540-44522-7. DOI : 10.1007/3-540-44522-6_3. URL : http://link.springer.com/10.1007/3-540-44522-6_3 (visité le 24/09/2018).

[33] Bertrand Daniel Dunay, Frederick E Petry et Bill P Buckles. « Regular language induction with genetic programming ». In : *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. IEEE. 1994, p. 396–400.

[34] Pierre Dupont. « Incremental regular inference ». en. In : *Grammatical Interference : Learning Syntax from Sentences*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, sept. 1996, p. 222–237. ISBN : 978-3-540-61778-5 978-3-540-70678-6. DOI : 10.1007/BFb0033357. URL : https://link.springer.com/chapter/10.1007/BFb0033357 (visité le 09/03/2018).

[35] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle. « Smews : Smart and mobile embedded web server ». In : *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on*. IEEE. 2009, p. 571–576.

[36] Isaac Evans et al. « Control Jujutsu : On the Weaknesses of Fine-Grained Control Flow Integrity ». In : *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA : ACM, 2015, p. 901–913. ISBN : 978-1-4503-3832-5. DOI : 10.1145/2810103.2813646. URL : http://doi.acm.org/10.1145/2810103.2813646 (visité le 18/02/2016).

[37] J Fenlason et R Stallman. « GNU gprof : The GNU profiler. Free Software Foundation ». In : *Inc. 52pp* (2000).

[38] Henning Fernau. « Learning XML Grammars ». In : *Machine Learning and Data Mining in Pattern Recognition*. Sous la dir. de G. Goos et al. T. 2123. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, p. 73–87. ISBN : 978-3-540-42359-1 978-3-540-44596-8. DOI : 10.1007/3-540-44596-X_7. URL : http://link.springer.com/10.1007/3-540-44596-X_7 (visité le 09/03/2018).

[39] Aurélien Francillon, Daniele Perito et Claude Castelluccia. « Defending Embedded Systems Against Control Flow Attacks ». In : *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*. SecuCode '09. New York, NY, USA : ACM, 2009, p. 19–26. ISBN : 978-1-60558-782-0. DOI : 10.1145/1655077.

1655083. URL : http://doi.acm.org/10.1145/1655077.1655083 (visité le 18/02/2016).

[40] Torsten FRENZEL et al. « Arm trustzone as a virtualization technique in embedded systems ». In : *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*. 2010, p. 29–42.

[41] P. GARCÍA, A. CANO et J. RUIZ. « A Comparative Study of Two Algorithms for Automata Identification ». en. In : *Grammatical Inference : Algorithms and Applications*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, sept. 2000, p. 115–126. ISBN : 978-3-540-41011-9 978-3-540-45257-7. DOI : 10.1007/978-3-540-45257-7_10. URL : https://link.springer.com/chapter/10.1007/978-3-540-45257-7_10 (visité le 09/03/2018).

[42] *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017*. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016.

[43] Xinyang GE et al. « Fine-Grained Control-Flow Integrity for Kernel Software ». In : (). URL : http://www.cse.psu.edu/~xxg113/eurosp16.pdf (visité le 18/02/2016).

[44] C Lee GILES et al. « Learning and extracting finite state automata with second-order recurrent neural networks ». In : *Neural Computation* 4.3 (1992), p. 393–405.

[45] Patrice GODEFROID, Adam KIEZUN et Michael Y LEVIN. « Grammar-based whitebox fuzzing ». In : *ACM Sigplan Notices*. T. 43. 6. ACM. 2008, p. 206–215.

[46] Patrice GODEFROID, Michael Y LEVIN et David MOLNAR. « SAGE : whitebox fuzzing for security testing ». In : *Queue* 10.1 (2012), p. 20.

[47] E. Mark GOLD. « Language identification in the limit ». In : *Information and control* 10.5 (1967), p. 447–474.

[48] *grsecurity*. URL : https://grsecurity.net/rap_faq.php (visité le 13/10/2018).

[49] Istvan HALLER et al. « Dowsing for overflows : a guided fuzzer to find buffer boundary violations. » In : *USENIX Security Symposium*. 2013, p. 49–64.

[50] William H. HAWKINS, Jason D. HISER et Jack W. DAVIDSON. « Dynamic Canary Randomization for Improved Software Security ». In : *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. CISRC '16. New York, NY, USA : ACM, 2016, 9 :1–9 :7. ISBN : 978-1-4503-3752-6. DOI : 10.1145/2897795.2897803. URL : http://doi.acm.org/10.1145/2897795.2897803 (visité le 13/10/2018).

[51] *HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack*. https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676.

[52] Hong HU et al. « Data-Oriented Programming : On the Expressiveness of Non-Control Data Attacks ». In : (2016). URL : http://www.ieee-security.org/TC/SP2016/papers/0824a969.pdf (visité le 13/07/2016).

[53] Faizan JAVED et al. « Context-free grammar induction using genetic programming ». In : *Proceedings of the 42nd annual Southeast regional conference*. ACM. 2004, p. 404–405.

[54] Karthick JAYARAMAN et al. « jFuzz : A concolic whitebox fuzzer for Java ». In : (2009).

[55] A. KANUPARTHI, J. RAJENDRAN et R. KARRI. « Controlling your control flow graph ». In : *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2016, p. 43–48. DOI : 10.1109/HST.2016.7495554.

[56] SOO-YOUNG KIM et GYUNGHO LEE. « Monitoring Control Flow History To Detect Code Reuse Attacks ». In : (). URL : http://www.wseas.us/e-library/conferences/2015/Seoul/ACE/ACE-14.pdf (visité le 18/02/2016).

[57] Dan KLEIN et Christopher D MANNING. « Natural language grammar induction using a constituent-context model ». In : *Advances in neural information processing systems*. 2002, p. 35–42.

[58] Emin Erkan KORKMAZ et Göktürk ÜÇOLUK. « Genetic programming for grammar induction ». In : (2001).

[59] Volodymyr KUZNETSOV et al. « Code-Pointer Integrity. » In :

[60] C. LATTNER et V. ADVE. « LLVM : a compilation framework for lifelong program analysis transformation ». In : *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Mar. 2004, p. 75–86. DOI : 10.1109/CGO.2004.1281665.

[61] Chris LATTNER. « LLVM and Clang : Next generation compiler technology ». In : *The BSD conference*. 2008, p. 1–2.

[62] Valentin LEFILS, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY. « EE-CFI : Externalized Control Flow Integrity for Embedded Devices ». In : *Innovative Mobile and Internet Services in Ubiquitous Computing* (2018).

[63] Valentin LEFILS, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY. « Model induction for better control flow integrity ». In : *To be published* (2018).

[64] Kyung-suk LHEE et Steve CHAPIN. « Buffer Overflow and Format String Overflow Vulnerabilities ». In : *Electrical Engineering and Computer Science* (jan. 2002). URL : http://surface.syr.edu/eecs/96.

[65] Zhuowei LI, A. DAS et Jianying ZHOU. « Theoretical basis for intrusion detection ». In : *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*. Juin 2005, p. 184–192. DOI : 10.1109/IAW.2005.1495951.

[66] Kangjie LU et al. « How to Make ASLR Win the Clone Wars : Runtime Re-Randomization ». In : (2016). URL : http://www.cc.gatech.edu/~klu38/publications/runtimeaslr-ndss16.pdf (visité le 18/02/2016).

[67] Ali Jose MASHTIZADEH et al. « CCFI : Cryptographically Enforced Control Flow Integrity ». In : *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA : ACM, 2015, p. 941–951. ISBN : 978-1-4503-3832-5. DOI : 10.1145/2810103.2813676. URL : http://doi.acm.org/10.1145/2810103.2813676 (visité le 18/02/2016).

[68] ARM MBED. « Welcome to Mbed ». In : *Disponible en ligne : https ://mbed.org* (2017).

[69] Scott MILLER et al. « Hidden understanding models of natural language ». In : *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 1994, p. 25–32.

[70] Vishwath MOHAN et al. « Opaque Control-Flow Integrity ». en. In : Internet Society, 2015. ISBN : 978-1-891562-38-9. DOI : 10.14722/ndss.2015.23271. URL : http://www.internetsociety.org/doc/opaque-control-flow-integrity (visité le 18/02/2016).

[71] Tilo MÜLLER. « ASLR smack & laugh reference ». In : *Seminar on Advanced Exploitation Techniques*. 2008. URL : https://www.cs.umd.edu/class/fall2015/cmsc414-0201/papers/aslr-smack-laugh.pdf (visité le 13/06/2016).

[72] Santosh NAGARAKATTE et al. « SoftBound : Highly compatible and complete spatial memory safety for C ». In : *ACM Sigplan Notices* 44.6 (2009), p. 245–258. URL : http://dl.acm.org/citation.cfm?id=1542504 (visité le 13/07/2016).

[73] Nicholas Nethercote et Julian Seward. « Valgrind : a framework for heavy-weight dynamic binary instrumentation ». In : *ACM Sigplan notices*. T. 42. 6. ACM. 2007, p. 89–100.

[74] Ben Niu. « Practical Control-Flow Integrity ». In : (2016). URL : http://www.cse.psu.edu/~gxt29/paper/BEN_NIU_Dissertation.pdf (visité le 13/07/2016).

[75] Ben Niu et Gang Tan. « Modular control-flow integrity ». In : *ACM SIGPLAN Notices* 49.6 (2014), p. 577–587.

[76] Ben Niu et Gang Tan. « Per-Input Control-Flow Integrity ». In : *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA : ACM, 2015, p. 914–926. ISBN : 978-1-4503-3832-5. DOI : 10.1145/2810103.2813644. URL : http://doi.acm.org/10.1145/2810103.2813644 (visité le 18/02/2016).

[77] Aleph One. « Smashing the stack for fun and profit ». In : *Phrack magazine* 7.49 (1996), p. 14–16. URL : http://www1.telhai.ac.il/sources/private/academic/cs/557/2659/Materials/Smashing.pdf (visité le 18/08/2016).

[78] Ulziibayar Otgonbaatar. « Evaluating Modern Defenses Against Control Flow Hijacking ». Thèse de doct. MIT Lincoln Laboratory, 2015. URL : http://people.csail.mit.edu/hes/ROP/Publications/ulzi-thesis.pdf (visité le 18/02/2016).

[79] Vasilis Pappas. « kBouncer : Efficient and transparent ROP mitigation ». In : *Apr* 1 (2012), p. 1–2. URL : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.383.261&rep=rep1&type=pdf (visité le 18/02/2016).

[80] Rajesh Parekh et Vasant Honavar. « Learning DFA from simple examples ». en. In : *Algorithmic Learning Theory*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, oct. 1997, p. 116–131. ISBN : 978-3-540-63577-2 978-3-540-69602-5. DOI : 10.1007/3-540-63577-7_39. URL : https://link.springer.com/chapter/10.1007/3-540-63577-7_39 (visité le 09/03/2018).

[81] Mathias Payer et Thomas R. Gross. « String Oriented Programming : When ASLR is Not Enough ». In : *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. PPREW '13. New York, NY, USA : ACM, 2013, 2 :1–2 :9. ISBN : 978-1-4503-1857-0. DOI : 10.1145/2430553.2430555. URL : http://doi.acm.org/10.1145/2430553.2430555 (visité le 18/02/2016).

[82] Aravind Prakash et Heng Yin. « Defeating ROP Through Denial of Stack Pivot ». In : *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC 2015. New York, NY, USA : ACM, 2015, p. 111–120. ISBN : 978-1-4503-3682-6. DOI : 10.1145/2818000.2818023. URL : http://doi.acm.org/10.1145/2818000.2818023 (visité le 18/02/2016).

[83] Pengfei Qiu et al. « Physical unclonable functions-based linear encryption against code reuse attacks ». en. In : ACM Press, 2016, p. 1–6. ISBN : 978-1-4503-4236-0. DOI : 10.1145/2897937.2898061. URL : http://dl.acm.org/citation.cfm?doid=2897937.2898061 (visité le 13/07/2016).

[84] Ganesan Ramalingam. « The undecidability of aliasing ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), p. 1467–1471. URL : http://dl.acm.org/citation.cfm?id=186041 (visité le 13/07/2016).

[85] Will Reese. « Nginx : the high-performance web server and reverse proxy ». In : *Linux Journal* 2008.173 (2008), p. 2.

[86] Andrs Riancho. « w3af-web application attack and audit framework ». In : *World Wide Web electronic publication* 21 (2011).

[87] Ryan Roemer et al. « Return-Oriented Programming : Systems, Languages, and Applications ». In : *ACM Trans. Inf. Syst. Secur.* 15.1 (mar. 2012), 2 :1–2 :34. ISSN :

1094-9224. DOI : 10.1145/2133375.2133377. URL : http://doi.acm.org/10.1145/2133375.2133377 (visité le 17/02/2016).

[88] *RTOS - Free professionally developed and robust real time operating system for small embedded systems development*. URL : http://www.freertos.org/RTOS.html (visité le 29/08/2016).

[89] Edward J. SCHWARTZ, Thanassis AVGERINOS et David BRUMLEY. « Q : Exploit Hardening Made Easy. » In : *USENIX Security Symposium*. 2011, p. 25–41. URL : http://static.usenix.org/legacy/events/sec11/tech/full_papers/Schwartz.pdf (visité le 19/04/2016).

[90] 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. New York, NY, USA : ACM, 2007, p. 552–561. ISBN : 978-1-59593-703-2. DOI : 10.1145/1315245.1315313. URL : http://doi.acm.org/10.1145/1315245.1315313 (visité le 13/10/2018).

[91] Hovav SHACHAM et al. « On the Effectiveness of Address-space Randomization ». In : *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. New York, NY, USA : ACM, 2004, p. 298–307. ISBN : 978-1-58113-961-7. DOI : 10.1145/1030083.1030124. URL : http://doi.acm.org/10.1145/1030083.1030124 (visité le 13/10/2018).

[92] Chengyu SONG et al. « Enforcing Kernel Security Invariants with Data Flow Integrity ». In : (2016). URL : http://www.cc.gatech.edu/grads/c/csong43/ndss16-kenali.pdf (visité le 18/02/2016).

[93] *St. Jude Medical drops after Muddy Water findings of 'negligent product design'*. https://www.cnbc.com/2016/08/25/st-jude-medical-drops-after-muddy-water-findings-of-negligent-product-design.html.

[94] Nick STEPHENS et al. « Driller : Augmenting Fuzzing Through Selective Symbolic Execution. » In : *NDSS*. T. 16. 2016, p. 1–16.

[95] *STM32F4 Series - STMicroelectronics*. URL : http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series.html?querycriteria=productId=SS1577 (visité le 29/08/2016).

[96] *Stuxnet Worm Attack on Iranian Nuclear Facilities*. http://large.stanford.edu/courses/2015/ph241/holloway1/.

[97] Michael SUTTON, Adam GREENE et Pedram AMINI. *Fuzzing : brute force vulnerability discovery*. Pearson Education, 2007.

[98] « Control Flow Integrity for COTS Binaries ». eng. In : sous la dir. de SYSTEMS ADMINISTRATION CONFERENCE et USENIX ASSOCIATION. Berkeley, Calif : USENIX Association, 2013. ISBN : 978-1-931971-03-4.

[99] Caroline TICE et al. « Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM ». en. In : (), p. 16.

[100] *U.S. security expert says surveillance cameras can be hacked*. https://www.reuters.com/article/us-surveilance-hackers-idUSBRE95G10520130617.

[101] Perry WAGLE, Crispin COWAN et al. « Stackguard : Simple stack smash protection for gcc ». In : *Proceedings of the GCC Developers Summit*. Citeseer. 2003, p. 243–255.

[102] D. WAGNER et D. DEAN. « Intrusion detection via static analysis ». In : *2001 IEEE Symposium on Security and Privacy, 2001. S P 2001. Proceedings*. 2001, p. 156–168. DOI : 10.1109/SECPRI.2001.924296.

[103] Z. WANG et X. JIANG. « HyperSafe : A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity ». In : *2010 IEEE Symposium on Security and Privacy*. Mai 2010, p. 380–395. DOI : 10.1109/SP.2010.30.

[104] Raymond L Watrous et Gary M Kuhn. « Induction of finite-state languages using second-order recurrent networks ». In : *Neural Computation* 4.3 (1992), p. 406–414.

[105] Reinhold P Weicker. « Dhrystone : a synthetic systems programming benchmark ». In : *Communications of the ACM* 27.10 (1984), p. 1013–1030.

[106] TA Welch. « A Technique for High-Performance Data Compression ». In : *Computer* 6 (1984), p. 8–19.

[107] En-Hui Yang et John C Kieffer. « Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models ». In : *IEEE Transactions on Information Theory* 46.3 (2000), p. 755–777.

[108] M.M. Yasin et A.A. Awan. « A study of host-based IDS using system calls ». In : *Networking and Communication Conference, 2004. INCC 2004. International*. Juin 2004, p. 36–41. doi : 10.1109/INCC.2004.1366573.

[109] Chao Zhang et al. « VTrust : Regaining Trust on Virtual Calls ». In : (2016). url : http://chao.100871.net/papers/VTrust-NDSS16-CR.pdf (visité le 18/02/2016).

[110] C. Zhang et al. « Practical Control Flow Integrity and Randomization for Binary Executables ». In : *2013 IEEE Symposium on Security and Privacy*. Mai 2013, p. 559–573. doi : 10.1109/SP.2013.44.

[111] Jacob Ziv et Abraham Lempel. « Compression of individual sequences via variable-rate coding ». In : *IEEE transactions on Information Theory* 24.5 (1978), p. 530–536.