Contrôle de flux d'information par utilisation conjointe d'analyse statique et dynamique accélérée matériellement

Mounir Nasr Allah

DOCTORAT BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

THESE DE DOCTORAT DE

CENTRALESUPELEC

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Mounir NASR ALLAH

Contrôle de flux d'information par utilisation conjointe d'analyse statique et dynamique accélérée matériellement

Thèse présentée et soutenue à Rennes, le 14 décembre 2020
Unité de recherche : UMR IRISA (CIDRE)
Thèse N° : 2020CSUP0007

Rapporteurs avant soutenance :

Karine HEYDEMANN *Maître de conférences, Sorbonne Université*
Lilian BOSSUET *Professeur, Université Jean-Monnet de Saint-Etienne*

Composition du Jury :

Présidente :	Laurence PIERRE	*Professeur, Université Grenoble Alpes*
Examinateurs :	Karine HEYDEMANN	*Maître de conférences, Sorbonne Université*
	Lilian BOSSUET	*Professeur, Université Jean-Monnet de Saint-Etienne*
	Guillaume HIET	*Professeur Associé, CentraleSupélec*
	Pascal COTRET	*Professeur Associé, ENSTA Bretagne*
Dir. de thèse :	Ludovic MÉ	*Adjoint au directeur scientifique, INRIA*

Remerciements

Cette thèse de doctorat a été une longue aventure semée d'embûches qui n'aurait pas pu aboutir sans la précieuse aide et le soutien de certaines personnes que je souhaite remercier. Je tiens tout d'abord à remercier mes parents pour l'éducation qu'ils m'ont donnée, leur amour et leur soutien inconditionnel qui m'ont permis de m'épanouir dans mes études. Je remercie ma femme, Amal GOUIA, pour sa patience et le soutien dont elle a fait preuve durant ces longues années de travail.

Je remercie Guillaume HIET, Professeur associé à CentraleSupélec, qui m'a encadré de façon exemplaire tout au long de cette thèse. Guillaume a toujours su se rendre disponible et être à l'écoute. Je ne pouvais pas espérer meilleur accompagnement. Je souhaite remercier Pascal COTRET, Professeur associé à l'ENSTA Bretagne, qui m'a également encadré et énormément aidé pour la partie FPGA. Je remercie mon directeur de thèse Ludovic MÉ, Adjoint au directeur scientifique à l'INRIA, pour son aide et sa bienveillance durant toutes ces années. Je remercie Guy GOGNIAT, Professeur à l'Université de Bretagne Sud et Vianney LAPÔTRE, Professeur associé à l'Université de Bretagne Sud pour l'encadrement et les différentes réunions constructives qui ont permis de faire mûrir ces travaux de thèse.

Je remercie Isabelle PUAUT, Professeur à l'Unversité de Rennes et Frédéric TRONEL, Professeur associé à CentraleSupélec, qui ont suivi la progression de mes travaux de thèse avec intérêt durant toutes ces années lors des différents comités de suivi de thèse.

Malgré les nombreuses contraintes liées à la crise sanitaire du Covid-19 et les différents problèmes techniques, ma soutenance de thèse s'est bien passée grâce à la bienveillance du jury. Ainsi, je remercie Laurence PIERRE, Professeur à l'Université Grenobles Alpes, qui a présidé ma soutenance. Je remercie Karine HEYDEMANN, Maitre de conférence à Sorbonne Université et Lilian BOSSUET, Professeur à l'Université Jean-Monnet de Saint-Étienne, d'avoir accepté de rapporter ma thèse.

Pour finir, je remercie mon ami Pierre MAZAUD pour ses nombreuses relectures et corrections de mon manuscrit.

Table des matières

Introduction

Depuis quelques années, les systèmes embarqués sont en plein essor, notamment avec l'arrivée des objets connectés dans les foyers. À l'inverse des ordinateurs classiques (serveur, portable ou station de travail), ces systèmes informatiques sont intégrés dans des objets dont la fonction principale ne se résume pas à celle d'un ordinateur, comme un four à micro-ondes, un téléphone ou un système de contrôle industriel. Ces systèmes embarqués ont souvent la contrainte principale de posséder des ressources limitées. Toutefois, cette définition est très générique et la frontière avec l'informatique classique est assez floue. Le monde des systèmes embarqués est vaste et comprend des systèmes de nature et de capacité très diverses. Nous pouvons typiquement distinguer deux catégories de systèmes embarqués :

— les systèmes qui n'utilisent pas de système d'exploitation (Operating System (OS)), ou un système d'exploitation temps-réel, et qui ont recours à un microprocesseur aux ressources limitées, souvent un microcontrôleur, par exemple un processeur de la famille ARM Cortex-M ;
— les systèmes qui utilisent un OS complexe (*rich OS*) et un microprocesseur plus puissant (*application processor*), par exemple un processeur de la famille ARM Cortex-A.

Dans cette thèse, nous nous intéressons à cette deuxième catégorie de systèmes qui comprend notamment les smartphones, les tablettes tactiles, les montres ou les télévisions connectées. Ces systèmes sont aujourd'hui omniprésents dans les foyers, mais ils peuvent également être utilisés dans des contextes critiques, notamment militaires [1].

Nous nous intéressons plus précisément à la sécurisation de ces systèmes, qui consiste à assurer les trois propriétés fondamentales que sont :

— la **confidentialité**, qui garantit que seuls les utilisateurs autorisés peuvent accéder aux informations ;
— l'**intégrité**, qui garantit que les informations ne peuvent être modifiées que par des utilisateurs autorisés.

1. Par exemple le *Android Team Awareness Kit* : https://afresearchlab.com/technology/information-technology/tactical-assault-kit-tak/

— la **disponibilité**, qui garantit que le système fonctionne et réalise sa mission sans dégradation de service.

Les besoins de sécurité peuvent être exprimés en combinant ces trois propriétés fondamentales sur les différents biens du système. L'objectif d'un attaquant est alors de réaliser une intrusion, en violant une ou plusieurs de ces propriétés. Il réalise pour cela une ou plusieurs attaques, exploitant des vulnérabilités, qui sont des défauts dans la protection du système. L'intrusion est avérée lorsqu'une de ces attaques réussit et qu'un besoin de sécurité n'est plus garanti.

La sécurité des systèmes embarqués doit être une priorité. En effet, même pour un usage domestique, ces systèmes détiennent en grande majorité des données personnelles de leurs utilisateurs. Comme tout système informatique, ils comportent en pratique de nombreuses vulnérabilités qui peuvent être exploitées par un attaquant, afin de compromettre la sécurité du système. Ces vulnérabilités sont notamment liées au processus de développement des applications, le développeur pouvant involontairement introduire des erreurs dans son code qui, parfois, se traduisent par des vulnérabilités.

La plupart des logiciels grand public utilisent des procédures de tests, de débogage et d'analyse statique afin de détecter le plus rapidement possible ces défauts avant leurs distributions. La détection de défauts à l'aide de ces outils reste cependant limitée, car elle dépend fortement des hypothèses fournies par les développeurs. Lorsqu'une faille est présente dans le code d'une application, mais reste encore inconnue du grand public, on parle alors de vulnérabilité « zero-day ».

Ces dernières années, plusieurs affaires d'espionnage ont fait utilisation de vulnérabilités « zero-day », notamment sur des smartphones, permettant à des attaquants de faire fuir des données personnelles. Par exemple, on soupçonne l'Arabie Saoudite d'avoir utilisé ce moyen pour pouvoir géolocaliser et assassiner l'opposant politique et journaliste Jamal Khashoggi [2]. Le patron d'Amazon, Jeff Bezos, aurait également fait les frais d'espionnage dans cette affaire entre mai 2018 et février 2019. Le rapport d'investigation numérique légale réalisé par la firme FTI Consulting [3] indique que, pendant presque un an, des informations personnelles lui ont ainsi été dérobées.

Ces affaires illustrent l'importance de la sécurisation de ces systèmes. Les vulnérabilités sont exploitées par des attaques de plus en plus sophistiquées, qui sont souvent liées à la cybercriminalité ainsi qu'à des activités militaires ou de renseignement. Ces attaques peuvent combiner différents types de com-

2. https://www.nytimes.com/2018/12/02/world/middleeast/saudi-khashoggi-spyware-israel.html

3. https://assets.documentcloud.org/documents/6668313/FTI-Report-into-Jeff-Bezos-Phone-Hack.pdf

portements malveillants et exploiter des vulnérabilités à différents niveaux. Il peut s'agir par exemple de détourner le flot d'exécution d'une application en exploitant une vulnérabilité de type *buffer overflow* ou d'accéder à des fichiers confidentiels en exploitant une vulnérabilité de type *directory traversal*.

Il est donc nécessaire de contrer ces attaques, en raison des vulnérabilités qui sont, en pratique, toujours présentes sur les systèmes. Il convient donc en premier lieu de déployer des mécanismes de sécurité **préventifs**, qui visent à empêcher l'exploitation des vulnérabilités ou limiter les effets de leur exploitation. Ainsi, de nombreuses approches préventives ont été proposées dans la littérature et certaines sont couramment déployées dans les systèmes. Par exemple, il est possible d'implémenter des vérifications dynamiques dans l'environnement d'exécution (comme c'est le cas pour l'environnement d'exécution d'Android), d'utiliser des mécanismes cryptographiques ou du contrôle d'accès.

Toutefois, en pratique, ces mécanismes préventifs ne sont pas suffisants. En effet, ils peuvent ne pas être appliqués systématiquement. Par exemple, les applications Android comportent du code natif qui n'est pas vérifié dynamiquement par l'environnement d'exécution Android, à la différence du bytecode issu des parties développées en Java ou en Kotlin. En outre, chaque approche préventive ne permet pas de se prémunir contre tous les types d'attaques. Par exemple, l'utilisation de Java permet certes de s'affranchir des attaques visant la gestion de la mémoire, notamment celles exploitant les *buffer overflow*, mais elle n'empêche pas l'exploitation d'erreurs dans les algorithmes implémentés, par exemple, l'exploitation d'un *directory traversal*.

Il est donc important de recourir également à des approches **réactives**. Pour ce type d'approche, il s'agit en premier lieu de détecter les attaques ou les intrusions en surveillant les systèmes, à l'exécution. Cette étape permet ensuite d'envoyer des alertes à des opérateurs de sécurité, de modifier le comportement du système, ou de mettre le système infecté en quarantaine.

Il existe deux types d'approches de détection d'intrusions :

— la détection de comportements malveillants (*misuse-based detection*), à base de signatures exprimant des motifs caractéristiques d'attaques ou d'intrusions ;

— la détection d'anomalies (*anomaly-based detection*), qui s'appuie sur une définition du comportement légale du système, et qui considère toute déviation par rapport à ce modèle comme une intrusion.

Les approches par signatures sont les plus utilisées, car elles sont simples à mettre en œuvre. Toutefois, elles ne permettent de détecter que des attaques connues et nécessitent une mise à jour régulière de la base de signatures. La détection d'anomalie permet, en théorie, de s'affranchir de cette description des attaques. Le problème majeur pour ce type d'approche consiste à définir

le comportement de référence. Une des solutions possibles consiste à définir ce comportement en s'appuyant sur une politique de sécurité, qui exprime explicitement les besoins de sécurité du système surveillé.

Dans cette thèse, nous proposons d'utiliser une approche de suivi dynamique des flux d'information, ou Dynamic Information Flow Tracking (DIFT) en anglais, afin de détecter des atteintes à la confidentialité et l'intégrité des données à différents niveaux (par exemple, le détournement du flot de contrôle d'une application ou des attaques de plus haut niveau comme la fuite du contenu d'un fichier confidentiel). Le DIFT consiste à :

1. attacher des **étiquettes** (appelées *label* ou *tag* dans la littérature) à des **conteneurs d'information** (par exemple des fichiers, des variables ou des registres) et de spécifier une **politique** de flux d'information, c'est-à-dire des relations entre les étiquettes définissant les flux légaux ;
2. **propager** ces étiquettes à l'exécution afin de refléter les flux d'information qui ont lieu durant l'exécution des programmes et **détecter** toute **violation de la politique**.

Le DIFT peut être implémenté à différents niveaux. Les approches implémentant le DIFT au niveau de l'OS [1, 2] sont des approches à gros grain qui ne suivent que les flux d'information résultant des appels système. Les conteneurs d'informations considérés sont des fichiers ou des pages mémoire. Dans ces approches, le moniteur est généralement implémenté au sein du noyau de l'OS, ce qui le protège des attaques exploitant des vulnérabilités dans les applications utilisateur. La spécification de la politique consiste alors à étiqueter les fichiers, ce qui peut être réalisé par un administrateur. Ces approches ont un faible surcoût à l'exécution mais elles souffrent de deux limitations majeures :

— Elles surapproximent le comportement interne des applications, ce qui peut conduire à de faux positifs ;
— Elles ne peuvent pas être utilisées pour détecter des attaques bas-niveau comme celles détournant le flot d'exécution. Ces dernières nécessitent de s'intéresser à des conteneurs d'information à grain fin, comme les registres du microprocesseur.

Les approches à grain fin [3, 4, 5, 6, 7, 8, 9] permettent de surveiller chaque instruction exécutée par une application. Elles permettent d'étiqueter des conteneurs à grain fin comme les registres ou les mots stockés en mémoire. Ces approches sont plus précises et peuvent détecter des attaques bas-niveau. Toutefois, elles présentent également des limitations :

— Les moniteurs DIFT implémentés de manière logicielle [3] ne sont pas isolés de leur cible, le code du moniteur étant tissé dans celui de

l'application ;
— Les approches implémentées matériellement ne peuvent gérer des conteneurs persistants comme les fichiers [4]. En outre, ces approches sont souvent invasives (le cœur du Central Processing Unit (CPU) est modifié pour propager et stocker les étiquettes) ce qui en limite l'adoption. En pratique, seuls des processeurs de type *softcore*, implémentés sur FPGA, peuvent être utilisés.

Cette thèse propose de répondre à la question suivante : **Est-il possible d'implémenter un moniteur de sécurité basé sur le suivi de flux d'information qui soit flexible, non invasif pour le matériel, qui prenne en compte les différentes couches du système et qui permette de détecter différentes classes d'attaques ?**

Cette thèse s'inscrit dans le projet de recherche collaboratif HardBlare [4], financé par le Labex CominLabs. L'objectif de ce projet est de proposer une approche générique de détection d'anomalies s'appuyant sur le DIFT, en combinant des aspects logiciels et matériels. L'originalité de ce projet réside dans les points suivants :

— Nous combinons une approche DIFT à grain fin avec l'étiquetage au niveau de l'OS, ce qui permet d'attacher des étiquettes aux fichiers. L'utilisateur final peut ainsi spécifier la politique de sécurité et sauvegarder les contextes de sécurité entre les redémarrages du système, les fichiers étant des conteneurs d'information persistants.
— Nous implémentons la propagation d'étiquettes dans un coprocesseur matériel pour limiter la dégradation des performances et isoler le moniteur. Contrairement aux autres approches matérielles, notre approche ne requiert **aucune modification du processeur principal**.
— L'isolation du moniteur dans un coprocesseur dédié créé un fossé sémantique entre le moniteur et le système surveillé. Le coprocesseur isolé doit extraire certaines informations du processeur principal pour en déduire le comportement de l'application surveillée. Réduire cet écart sans modifier le CPU principal est l'un des principaux défis du projet. Nous nous appuyons pour cela sur les mécanismes de traces du microprocesseur.

Dans le cadre de ce projet, ma contribution a porté essentiellement sur les aspects logiciels. Plus précisément, je me suis attaché à résoudre le problème du fossé sémantique en proposant une combinaison originale de différentes approches :

— J'ai proposé de précalculer les **annotations** lors de la compilation des applications. Ces annotations reflètent les flux d'information dans

4. https://project.inria.fr/hardblare/

chaque bloc de base ;
- J'ai proposé une phase d'instrumentation de l'application permettant, lors des accès mémoire réalisés par l'application, d'envoyer les adresses cibles au moniteur.
- J'ai proposé un mécanisme de coopération entre le coprocesseur implémentant le moniteur DIFT et l'OS, afin de propager les étiquettes aux fichiers.

L'organisation de ce manuscrit est la suivante. Le chapitre 1 présente l'état de l'art en introduisant les différentes approches utilisées pour réaliser du contrôle de flux d'information de façon statique, dynamique, ou hybride dans les différentes couches du système. Le chapitre 2 détaille l'approche que nous proposons, en commençant par la mise à profit des composants de débogage et de traces afin de récupérer de façon non invasive des informations sur l'état d'exécution du processeur. Nous identifions ensuite les différents conteneurs d'information et les flux d'information présents dans la couche matérielle qui serviront à la propagation des étiquettes à l'aide d'annotations. Par la suite, nous expliquons comment envoyer de façon dynamique certaines informations manquantes au moniteur de sécurité à l'aide de l'instrumentation et d'un canal de communication avec le système d'exploitation. Pour finir, nous présentons l'architecture du moniteur de sécurité. Dans le chapitre 3, nous présentons l'élaboration de différentes politiques de sécurité permettant de détecter l'exploitation de certaines vulnérabilités ainsi que les résultats obtenus. Le chapitre 4 fait place à la conclusion de ces travaux et expose les perspectives envisagées.

Chapitre 1

État de l'art

Dans un système ou une organisation, il est parfois nécessaire de protéger certaines données ou ressources. En effet, leurs utilisations à des fins malveillantes pourraient entraîner des dommages économiques, politiques, légaux ou personnels. Pour garantir la sécurité de tout système, trois phases clés sont nécessaires :

La spécification d'une politique de sécurité, qui permet de décrire ce qui est autorisé ou interdit. C'est une description qui peut être exprimée de façon formelle ou informelle et qui n'est pas forcément dépendante de son implémentation. L'expression de politiques de sécurité peut par exemple prendre les formes suivantes :

- Seuls les salariés membres de l'équipe de recherche et développement peuvent accéder au laboratoire.
- Toute information médicale d'un patient peut être consultée uniquement par un médecin.

L'implémentation d'un mécanisme de sécurité, qui permet la mise en œuvre de la politique de sécurité. Un mécanisme de sécurité est donc la matérialisation d'une politique de sécurité et peut par exemple être :

- Une serrure électronique fonctionnant avec des badges de radio-identification qui permet de contrôler l'accès à un laboratoire selon l'identité d'une personne.
- Un mécanisme de contrôle d'accès protégeant l'accès à des fichiers sensibles selon le statut de l'utilisateur.

La vérification de l'implémentation, qui permet de s'assurer que l'implémentation est correcte et conforme à la spécification. Cette vérification peut être réalisée à l'aide d'outils, de processus ou de méthodes formelles.

— Une serrure de porte blindée peut par exemple être certifiée par le centre national de prévention et de protection afin de garantir une résistance minimale de quinze minutes en cas de tentative d'effraction par des cambrioleurs expérimentés dotés d'outils professionnels.
— Un mécanisme de contrôle d'accès dans une application peut par exemple être développé à l'aide d'un langage de programmation dont la sémantique permet de prouver facilement que le mécanisme est correct.

Dans cette thèse, nous nous intéresserons principalement à l'implémentation d'un mécanisme de sécurité au sein d'un ordinateur qui puisse permettre de mettre en œuvre différentes politiques de sécurité afin de protéger les applications et les données des utilisateurs. On se place dans un contexte où l'ordinateur héberge des données de différentes sensibilités et permet d'exécuter des programmes de différents niveaux de confiance. On cherche donc à se prémunir d'attaques logicielles où l'attaquant tente d'accéder à des données auxquelles il ne devrait pas avoir accès, et ainsi atteindre à leur confidentialité ou leur intégrité, en exploitant des vulnérabilités présentes dans les applications du système.

Historiquement, deux types de politiques de sécurité ont été proposées afin de garantir l'intégrité et la confidentialité des données au sein d'un ordinateur : les politiques de contrôle d'accès et les politiques de contrôle de flux d'information. Nous présentons ces deux types de politiques en section 1.1 et nous justifions la mise en œuvre de politiques de contrôle de flux en identifiant les limites des politiques de contrôle d'accès. Par la suite, nous nous intéressons aux mécanismes de contrôle de flux d'information (section 1.2) et nous présentons différentes stratégies afin d'implémenter un mécanisme de contrôle de flux d'information au sein d'un ordinateur de manière statique (section 1.3), dynamique (section 1.4) ou hybride (section 1.5).

1.1 Politiques de contrôle d'accès et politiques de contrôle de flux d'information

1.1.1 Politiques de contrôle d'accès

Parmi les mécanismes de sécurité, le contrôle d'accès est l'une des solutions les plus répandues dans les systèmes d'exploitation avec une première apparition en 1965 dans le système d'exploitation Multics [10, 11], suivi quelques années plus tard par les systèmes d'exploitation Unix, Linux

et Windows. Le contrôle d'accès permet de vérifier en amont de toute action sur un objet ou une ressource que l'utilisateur à l'origine de la requête d'accès est habilité à réaliser cette action.

Un mécanisme de contrôle d'accès repose sur un ***moniteur de référence*** qui étudie la requête envoyée par le sujet et qui autorise ou refuse l'accès à cet objet. Ce moniteur de référence peut-être incorporé dans les différentes couches du système. En effet, le contrôle d'accès peut être implémenté dans une application, dans un intergiciel, dans un système d'exploitation, ou dans des composants matériels. L'expression d'une politique de contrôle d'accès s'appuie donc sur les entités suivantes :

Un objet ou une ressource qui permet de recevoir, d'envoyer, ou de stocker de l'information.

Un sujet qui est l'identité de l'utilisateur ou de l'application qui souhaite accomplir une action sur un objet ou une ressource.

Un ensemble de droits d'accès qui correspond aux actions que le sujet est autorisé à mener sur un objet ou une ressource. Par exemple : la lecture, l'écriture, la modification, la suppression, etc.

La Figure 1.1 illustre le fonctionnement d'un mécanisme de contrôle d'accès, où l'utilisatrice Eve essaye d'écrire dans le fichier *notes.txt* sans avoir les droits requis : sa requête est donc refusée par le moniteur de référence. Elle est par contre autorisée à lire le fichier *fuite.txt*, car elle détient le droit de lire ce fichier.

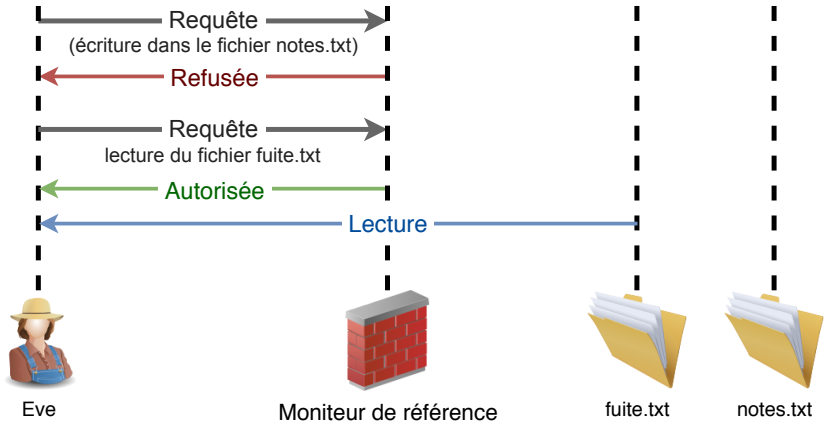


FIGURE 1.1 – Principe d'un mécanisme de contrôle d'accès.

En 1974, LAMPSON propose de représenter une politique de contrôle d'accès grâce à une matrice A impliquant les trois entités précédemment

citées [12]. Les lignes de la matrice désignent les sujets, les colonnes déterminent les objets ou les ressources. L'élément $A[i, j]$ décrit les droits d'accès que le sujet i possède sur l'objet j. Ces droits d'accès représentent les types d'actions que l'utilisateur est autorisé à effectuer sur le fichier, par exemple : la lecture, l'écriture.

Pour illustrer les limitations du contrôle d'accès, supposons un système contenant un fichier *notes.txt* qu'*Alice* peut lire et modifier, un fichier *fuite.txt* que *Bob* peut lire et modifier et un utilisateur *Eve* qui souhaite accéder aux informations contenues dans le fichier *notes.txt* de *Alice*. La matrice 1.1 présente les droits d'accès pour chaque sujet et objet du système.

	notes.txt	**fuite.txt**
Alice	{Lecture ; Écriture}	∅
Bob	{Lecture}	{Lecture ; Écriture}
Eve	∅	{Lecture}

TABLE 1.1 – Matrice de contrôle d'accès pour les fichiers *notes.txt* et *fuite.txt*.

Dans cet exemple illustré par la Figure 1.2, *Bob* peut lire le contenu du fichier *notes.txt*, contrairement à *Eve*. Cependant, rien n'empêche *Bob* de copier les données lues depuis le fichier *notes.txt* vers le fichier *fuite.txt*, et permettre à *Eve* de lire ce fichier. Une fuite d'information est donc occasionnée malgré le souhait d'interdire à *Eve* d'avoir accès aux données du fichier *notes.txt*.

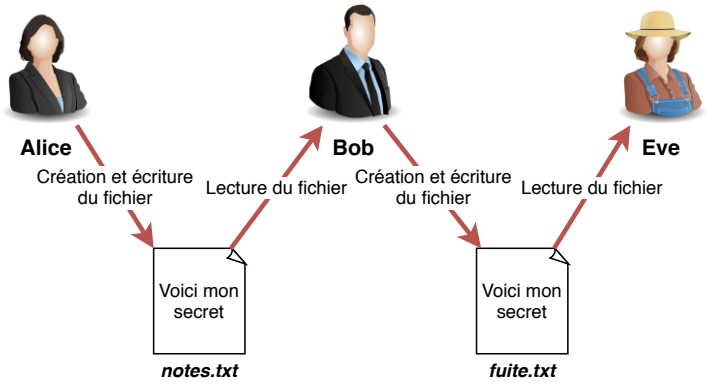


FIGURE 1.2 – Exemple de fuite de donnée possible avec un mécanisme de contrôle d'accès.

La question de la définition (et de la modification) des droits d'accès pour un objet est donc un élément déterminant dans la sécurité des modèles de

contrôle d'accès qui se divisent principalement en deux catégories distinctes : le contrôle d'accès discrétionnaire Discretionary access control (DAC) et le contrôle d'accès obligatoire Mandatory access control (MAC).

Le contrôle d'accès discrétionnaire (DAC) autorise la modification de la politique de contrôle d'accès d'un objet uniquement à son propriétaire. La notion de propriétaire est modélisée par un droit d'accès spécifique que possède un sujet sur un objet. Généralement, le propriétaire initial est le créateur de l'objet. Par exemple, le propriétaire du fichier *notes.txt* a le privilège de pouvoir modifier à sa guise les autorisations accordées aux autres utilisateurs. C'est donc un modèle décentralisé puisque chaque utilisateur a la capacité de modifier les permissions de ses propres fichiers. Il est alors difficile pour un administrateur d'avoir une vision globale de la sécurité du système.

Le contrôle d'accès obligatoire (MAC) ne donne pas de privilège au propriétaire d'un objet, seul l'administrateur du moniteur de référence a la capacité de modifier les permissions d'un objet. C'est donc un système centralisé dans lequel une autorité de confiance initie et modifie les permissions dans le système.

Les besoins du gouvernement des États-Unis d'Amérique en termes de confidentialité de l'information ont conduit le U.S. Department of Defense (DoD) à créer un modèle de politique de contrôle d'accès obligatoire multiniveau. Conçu par Bell et La Padula [13, 11], le modèle est basé sur une classification multiniveaux permettant le contrôle d'accès aux informations selon l'accréditation de l'utilisateur. Chaque utilisateur (ou sujet) possède un niveau d'habilitation et chaque objet est associé à un label de sécurité. Ces niveaux d'habilitation et labels de sécurité sont ordonnés. Par exemple, en France, les niveaux d'habilitation et de classification pour la protection du secret défense sont ordonnés de la manière suivante : « Très Secret Défense » $>$ « Secret Défense » $>$ « Confidentiel Défense ». Ce type de politique vise à s'assurer que les informations de niveau x puissent être lues uniquement par des utilisateurs accrédités à un niveau y, avec $y \geqslant x$. En pratique, pour implémenter ce type de politique au sein d'un ordinateur, il faut vérifier deux règles :

— Un sujet de niveau s peut lire les objets de niveau o si et seulement si $s \geqslant o$ (*read down* ou *Simple Security Property*)
— Un sujet de niveau s peut modifier les objets de niveau o si et seulement si $s \leqslant o$ (*write up* ou *Start Security Property*)

La première règle paraît assez naturelle. La seconde, moins intuitive, permet de s'assurer qu'un sujet malveillant, ou compromis, ne pourra en aucune

manière provoquer une fuite d'information vers un niveau inférieur. Ce type de modèle permet donc de se prémunir des attaques illustrées par la figure 1.2.

Le modèle de Bell-LaPadula garantit avant tout la confidentialité des informations. En effet, ce modèle autorise un utilisateur à lire uniquement les informations contenues dans les objets ou ressources ayant un niveau de sécurité inférieure ou égale à son niveau d'accréditation. L'intégrité des informations n'est malheureusement pas assurée, car ce même utilisateur peut écrire dans n'importe quel objet ayant une classe de sécurité supérieure ou égale à son accréditation. Le modèle dual proposé par Biba [14] permet d'assurer l'intégrité des données.

Ces modèles de politique de contrôle d'accès multiniveaux permettent de prévenir toute fuite ou modification d'information par un sujet qui ne possède pas le niveau d'accréditation nécessaire pour lire ou modifier l'information en question. Cependant, ces modèles sont en pratique peu utilisés, car ils sont très restrictifs. Par exemple, dans le modèle de Bell-LaPadula, un sujet ne peut simultanément accéder en lecture à des objets de niveau n et en écriture à des objets de niveau $m < n$, même si son exécution ne provoque aucune fuite d'information du niveau n vers le niveau m. Cette restriction limite en pratique l'usage de ces modèles à des contextes très spécifiques, typiquement dans des usages militaires ou gouvernementaux. Toutefois, la plupart des usages s'accommodent mal de ces règles contraignantes. Ainsi, un utilisateur souhaite généralement exécuter simultanément (c'est-à-dire dans la même session) différentes applications (client SSH, navigateur Web, client de courriel, etc.) et que ces applications puissent lire et modifier leurs fichiers. Pour autant, on souhaite s'assurer qu'une application malveillante ou compromise ne pourra atteindre à la confidentialité ou l'intégrité des données d'une autre application.

Bien que très répandu, le contrôle d'accès permet de mener des vérifications uniquement avant l'accès à un objet. Une fois l'autorisation d'accéder à l'objet accordée, la propagation de ces données dans le système échappe totalement au contrôle d'accès. Cela peut conduire à des fuites d'information sans violation apparente de la politique de contrôle d'accès. Pour se prémunir contre de telles attaques, il est nécessaire de recourir à des modèles de contrôle d'accès obligatoire, qui sont en pratique trop restrictifs.

Pour pouvoir régler le problème de fuite d'information et contrôler de manière précise la propagation des données après leur accès, il est donc essentiel de raisonner en termes de flux d'information et de vérifier que toute propagation d'une information respecte la politique de sécurité voulue. Ce mécanisme est connu sous le nom de « contrôle de flux d'information », Information Flow Control (IFC) en anglais.

1.1.2 Politiques de contrôle de flux d'information

Dans les années 1970, plusieurs chercheurs ont commencé à se pencher sur les flux d'information. Fenton [15] s'est d'abord intéressé au développement d'un système permettant de se prémunir des fuites d'information grâce au marquage des données avec un niveau de sécurité. Pour les flux d'information, nous parlerons de ***conteneurs d'information*** pour décrire les objets abstraits qui permettent de stocker de l'information. Un flux d'information se produit donc lorsqu'une information est propagée entre un conteneur d'information source A et un conteneur de destination B.

Afin de définir plus formellement cette notion de propagation de l'information, plusieurs propriétés ont été proposées dans la littérature. Goguen et Meseguer [16] ont énoncé proposé la propriété de non-interférence qui a largement été adoptée par la communauté. Cette propriété permet de s'assurer que plusieurs exécutions d'un programme dont seules les données sensibles ont été modifiées produisent les mêmes données publiques, visibles par un attaquant. Cette propriété illustrée par la Figure 1.3, permet donc de se prémunir de la fuite de données en s'assurant qu'aucune information possédant une classe de sécurité donnée, n'influence de quelque façon que ce soit des données ayant un niveau de sécurité inférieur. Historiquement, cette propriété a été proposée pour garantir la confidentialité des données. Toutefois, elle peut également être utilisée de manière duale pour garantir l'intégrité des données en s'assurant qu'un attaquant ne peut, à partir d'entrée publique qu'il peut modifier, influencer la valeur de certains conteneurs.

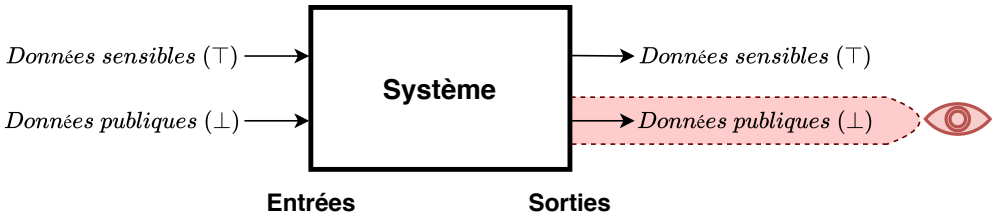


Figure 1.3 – Principe de non-interférence.

Cette propriété est très générique, mais elle est aussi très restrictive. En effet, certains programmes ne respectent pas cette propriété sans pour autant permettre à un attaquant d'inférer suffisamment d'information sur les entrées secrètes à partir de l'observation des sorties publiques. De manière générale, elle ne permet pas de quantifier la fuite d'information mais donne des conditions restrictives qui garantissent l'absence de fuite. Par exemple, l'utilisation de fonctions cryptographiques permet de se prémunir contre la fuite d'information mais donne généralement des programmes qui ne vérifient pas, au

sens strict, la non-interférence. Ainsi, si l'on suppose une fonction h permettant de calculer un résumé cryptographique (par exemple SHA-256), alors le programme $x = h(y)$, avec y une variable secrète et x une variable publique, ne vérifie pas la propriété de non-interférence. En effet, différentes exécutions de ce programme pour différentes valeurs de y (secrètes) conduiront à différentes valeurs (publiques) de x, observables par un attaquant. Toutefois, les fonctions de calcul de résumés cryptographiques sont construites de manière à ce que cette variation des valeurs de x ne permettent pas à un attaquant, à l'échelle humaine, d'inférer de l'information sur la valeur de y. Afin de prendre en compte ces différents cas de figure où l'interférence est acceptable, il est nécessaire de recourir à la déclassification, qui permet de définir des exceptions à la propriété de non-interférence [17].

Cette propriété, telle qu'elle est définie ici, ne s'intéresse qu'au cas où l'attaquant peut inférer un flux d'information uniquement en observant la valeur du conteneur de destination. Toutefois, l'exécution des programmes peut conduire à des effets de bords qui vont entraîner la fluctuation d'autres grandeurs physiques (par exemple, la consommation énergétique, les émissions électromagnétiques ou les temps d'exécution), observables par l'attaquant (sans nécessairement changer la valeur des conteneurs publics). On parle alors d'attaques par canaux cachés ou par canaux auxiliaires. On remarque qu'il est possible d'étendre la notion de non-interférence pour prendre en compte ces attaques [18, 19], en considérant l'observation de ces grandeurs physiques comme des conteneurs d'information publique, observable par l'attaquant. Ainsi, en définissant le pouvoir d'observation de l'attaquant, on définit les classes d'attaques considérées.

Dans cette thèse, on se restreint au cas fréquemment considéré dans la littérature où l'attaquant ne peut observer que les valeurs des conteneurs, ce qui correspond à une large classe d'attaques logicielles, dont on cherche à se protéger. Cela permet par exemple de se prémunir contre les attaques exploitant des vulnérabilités dans la gestion de la mémoire [7] (*buffer overflow, format string attack*, etc.), l'injection de code [20] (XSS, SQL injection, etc.) ou les *directory traversal* [21]. Ces différentes attaques correspondent en pratique aux classes de vulnérabilités les plus courantes, selon le classement établi en 2020 par le MITRE [22]. Dans ce contexte, il existe deux types de flux d'information selon la façon dont l'information contenue dans un conteneur source A influe sur l'information contenue dans le conteneur destination B.

Les flux explicites se produisent lorsqu'une information provenant d'un conteneur source est utilisée via une expression pour générer une nouvelle valeur affectée à un conteneur destination. Par exemple, lors de

l'affectation dans le code 1.1, un flux d'information explicite se produit entre la variable a et b.

```
1 b = a;
```

CODE SOURCE 1.1 – Exemple d'un flux d'information explicite

Un flux implicite se produit lorsque des informations contenues dans un conteneur conditionnent des flux d'information explicites qui se produisent dans d'autres conteneurs. Le code 1.2 contient un branchement conditionnel sur la valeur de la variable *motdepasse*. L'attribution des valeurs *true* ou $false$ à la variable *auth* dépend donc implicitement de la valeur contenue dans la variable *motdepasse*. En effet, si un attaquant a la possibilité d'observer la valeur de la variable *auth*, il pourra alors en déduire des informations sur la valeur de la variable *motdepasse*. Il y a donc un flux implicite du conteneur *motdepasse* vers le conteneur *auth*.

```
1 if(motdepasse == 1234) {
2  auth = true;
3 }
4 else {
5  auth = false;
6 }
```

CODE SOURCE 1.2 – Exemple de flux d'information implicite

Une politique de contrôle de flux d'information consiste à définir des niveaux de sécurité et les flux d'information autorisés entre ces niveaux. Ces niveaux seront ensuite utilisés pour marquer les conteneurs et définir les flux autorisés entre conteneurs d'information.

Les niveaux de sécurité peuvent correspondre à une propriété d'intégrité ou de confidentialité. Ces niveaux de sécurité peuvent être représentés sous une forme d'ordre linéaire (Figure 1.4), ou sous une forme plus complexe en utilisant des ordres partiels (Figure 1.5).

Dans plusieurs travaux [15, 3, 23, 24, 25], la politique de flux d'information se base uniquement sur deux niveaux de sécurité, $\top$ et $\bot$.

$\top$ représente soit un niveau de sécurité confidentiel soit une donnée authentique provenant d'une source de confiance.

$\bot$ représente soit un niveau de sécurité public soit une donnée provenant d'une source potentiellement malveillante.

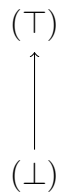


FIGURE 1.4 – Représentation avec deux niveaux de sécurité.

La représentation à deux niveaux de sécurité illustrée par la Figure 1.4, est très limitante et ne laisse qu'une vision binaire en terme de sécurité à l'administrateur du système. La solution multiniveau proposé par DENNING [26], permet de décrire les politiques de flux d'information grâce à une structure algébrique appelée treillis qui est beaucoup plus flexible.

Le modèle de politique de flux d'information en treillis est défini de la façon suivante :

$$\langle SC, \Rightarrow, \oplus, \otimes, \top, \bot \rangle \tag{1.1}$$

Où :

$SC = [s_1, s_2, s_3, \dots]$ est un ensemble de niveaux de sécurité

$\Rightarrow$ est une relation d'ordre sur les niveaux de sécurité qui permet de savoir si un flux d'information entre deux niveaux de sécurité est autorisé ou non.

$\oplus$ est l'opérateur permettant d'obtenir le niveau de sécurité dominant entre plusieurs niveaux de sécurité. Lors d'un mélange d'informations entre plusieurs conteneurs, on utilise cet opérateur.

$\otimes$ est l'opérateur permettant de récupérer le niveau de sécurité inférieure entre plusieurs niveaux de sécurité.

$\top$ représente le plus grand niveau de sécurité, autrement dit la borne supérieure.

$\bot$ représente le plus petit niveau de sécurité, autrement dit la borne inférieure.

La Figure 1.5 donne un exemple des niveaux de sécurité qui peuvent être utilisés pour classifier les données sensibles d'un pays.

Dans un système informatique complet, des flux d'information se produisent dans plusieurs couches : entre les structures de donnée et les fichiers du système d'exploitation, entre les variables et la mémoire d'une application, mais également au niveau du matériel entre les registres du processeur.

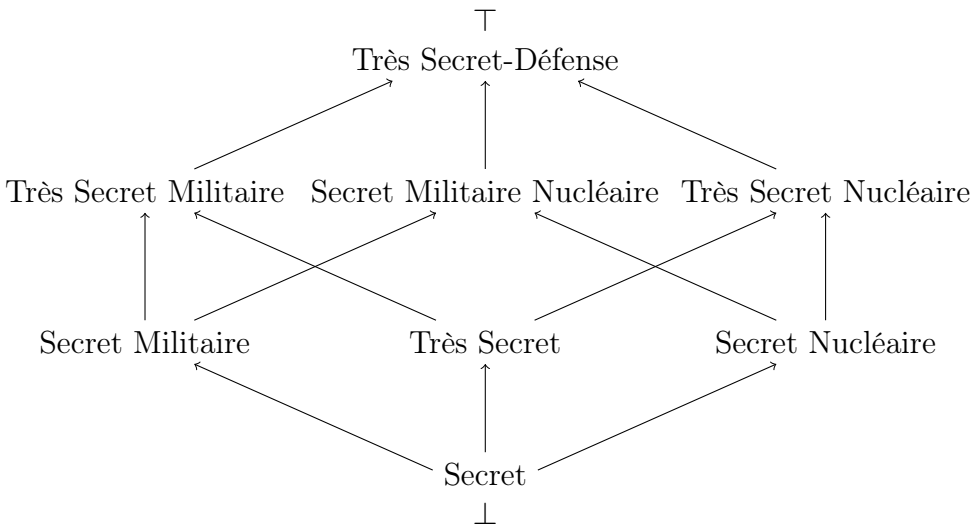


FIGURE 1.5 – Exemple de niveaux de sécurité en treillis.

La Figure 1.6 illustre les flux d'information qui se produisent lors de l'exécution d'un processus P qui a pour but l'addition du contenu de deux fichiers. Pour des raisons de simplification, les niveaux de sécurité sont ici représentés par des couleurs. Ce programme ouvre les fichiers A, B et C, qui ont initialement été marqués par des niveaux de sécurité jaune, bleu et violet. Le programme charge dans trois zones mémoires différentes (zone mémoire 1, 2 et 3) les informations contenues respectivement dans les fichiers A, B et C. Une fois ces informations chargées en mémoire, le processus souhaite additionner les données contenues dans la zone mémoire 1 et 2. Pour que le CPU puisse additionner les informations de la zone mémoire 1 et 2, les données doivent d'abord être chargées dans les registres du processeur. Ainsi, une fois l'addition effectuée, les niveaux de sécurité des registres $R2$ et $R3$ sont combinés pour donner un niveau de sécurité de couleur verte, qui est ensuite propagé au registre $R4$.

Puisque cette thèse s'intéresse au contrôle de flux d'information dans toutes les couches du système, nous regrouperons tous les objets cités précédemment par la notion de conteneurs d'information. Dans cet exemple, les fichiers peuvent contenir plusieurs gigaoctets de données : être contraint de marquer tout un fichier avec un seul niveau de sécurité introduit une imprécision. Une zone mémoire fait en général quelques mégaoctets. Lorsque l'on atteint les registres CPU, il est alors possible de marquer un octet ou un mot de 32 bits avec un niveau de sécurité, ce qui constitue la plus petite taille dans la plupart des machines modernes. Nous définissons donc la granularité comme la taille du conteneur d'information qui doit être marqué avec un

niveau de sécurité :

Gros grain : est la granularité pour des fichiers.

Grain moyen : est la granularité pour des zones mémoires.

Grain fin : pour les registres du processeur.

Il y a donc un compromis à faire entre la granularité utilisée et les ressources allouées pour permettre le contrôle de flux d'information. En effet, plus la granularité est petite, plus on doit propager et stocker un nombre important de labels de sécurité.

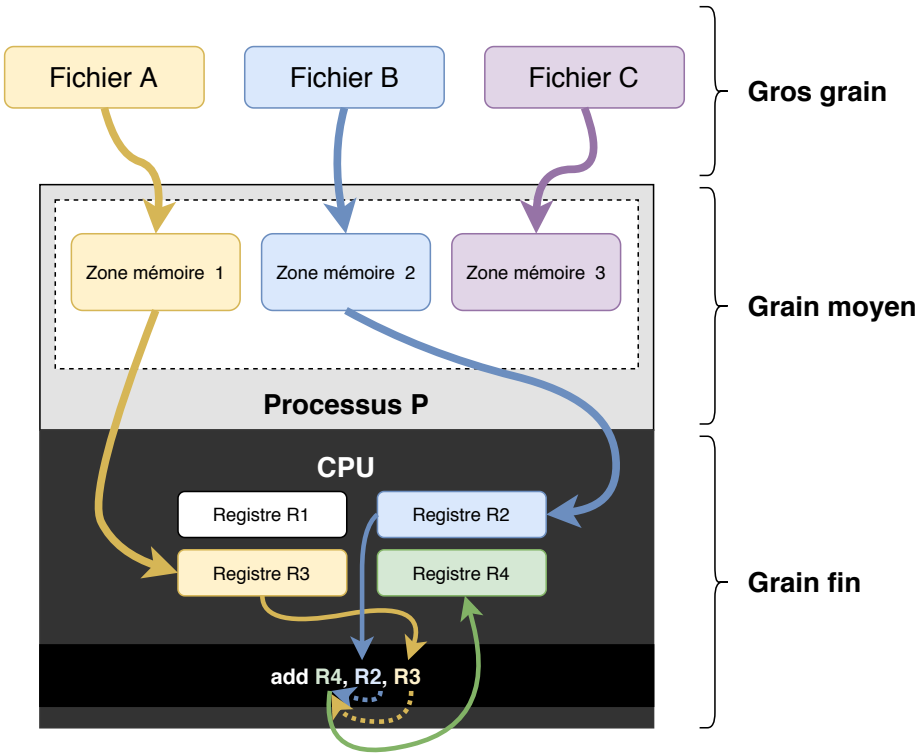


FIGURE 1.6 – Principe de la propagation de teintes.

Les politiques de suivi de flux d'informations se basent sur un socle théorique et mathématique robuste. Malheureusement, leur application est très complexe car les flux d'informations se produisent dans les différentes couches du système. Il est donc nécessaire d'avoir un mécanisme de sécurité appliquant la politique de flux d'information dans chaque couche.

Les premiers travaux ont tenté de résoudre le problème en proposant une nouvelle architecture machine incluant directement les différentes opérations nécessaires au contrôle de flux d'information et fera l'objet de la section 1.2.

Cette approche est cependant restée théorique car elle implique une modification profonde de l'architecture et une fixation de la politique de sécurité à seulement deux niveaux de sécurité. Par la suite, des travaux (présentés en section 1.3) se sont intéressés à vérifier statiquement le respect d'une politique de flux d'information. Ce type d'approches est cependant très restrictive et permet difficilement de prendre en compte les aspects dynamiques d'un système d'information. Des approches dynamiques, présentées en section 1.4 ont également été proposées pour contrôler les flux d'informations.

1.2 Mécanisme de contrôle de flux d'information vu comme une machine abstraite

Les flux d'information ont d'abord été utilisés pour résoudre le problème du confinement énoncé par Lampson [27] : comment confiner un programme potentiellement vulnérable durant son exécution pour que l'application ne puisse en aucun cas faire fuir de l'information censée être confidentielle ?

Au vu de la difficulté d'implémenter une solution, Fenton [15] propose la création d'une machine abstraite adaptée pour le IFC, basée sur une machine à compteur aussi connu sous le nom de Machine de Minsky [28]. Cet automate à états finis contient des registres et trois instructions rudimentaires décrites dans le tableau 1.2.

Instruction	**Sémantique**	**Description**
a'	`a = a + 1;`	Incrémenter le registre a de un
$a^-(n)$	`if(a == 0) goto n; else a = a - 1;`	Si le registre a est égal à zéro se rendre à l'état n, sinon décrémenter a de un
halt	`exit;`	Terminer un programme

Table 1.2 – Machine de Minsky.

Fenton étend ce modèle en modifiant la sémantique des instructions, en ajoutant deux nouvelles instructions et en attachant à chaque registre un niveau de sécurité qui ne peut prendre que deux niveaux de sécurité : **priv** pour représenter une donnée confidentielle, et **null** pour représenter une donnée publique.

En effet, pour pouvoir protéger l'exécution arbitraire d'un programme sur lequel l'utilisateur peut contrôler le chemin pris dans le graphe de flot

de contrôle, et respecter la propriété de non-interférence, les flux implicites doivent être pris en considération.

Ainsi, chaque donnée dans la machine abstraite est caractérisée par un niveau de sécurité, y compris pour le registre de compteur ordinal appelé registre Program Counter (PC). La position dans un programme contient une quantité d'informations importantes qui peuvent fuiter sur les canaux auxiliaires. C'est pourquoi le registre PC doit être marqué comme tout autre conteneur d'information dans la machine. Ainsi p est le registre PC et $\underline{p}$ symbolise son niveau de sécurité. Marquer le registre PC avec un niveau de sécurité permet donc de prendre en considération les flux d'information implicites.

Grâce à cette solution, quatre théorèmes sont vérifiés :

Théorème 1 : le système est sécurisé si et seulement si p est marqué avec un niveau de sécurité *null* et ne dépend en aucun cas d'une information marquée *priv*.

Théorème 2 : si le chemin d'exécution pris (registre PC) a un niveau de sécurité *priv*, alors il est impossible de modifier une variable marquée *null*.

Théorème 3 : un chemin d'exécution (registre PC) ayant pour niveau de sécurité *null* ne peut en aucun cas exécuter une instruction de branchement dépendant d'une information ayant un niveau de sécurité *priv* sans que p ne soit lui-même marqué avec le niveau de sécurité *priv*.

Théorème 4 : si $\underline{p} = priv$ le seul moyen pour que le niveau du registre PC soit remis à *null* est en exécutant l'instruction *Return*

Bien que cette approche puisse garantir de fortes garanties de sécurité et notamment la non-interférence, elle reste purement théorique et nécessite la modification de toute l'architecture de la machine pour pouvoir stocker les niveaux de sécurité et insérer toute la logique d'exécution des instructions.

Les chercheurs se sont donc penchés sur des mécanismes de contrôle de flux d'information concrets. Deux approches se sont alors distinguées : les mécanismes statiques qui sont appliqués avant l'exécution de l'application grâce à une analyse statique, et les mécanismes dynamiques qui interviennent en même temps que son exécution.

Les solutions dynamiques sont basées sur la vérification d'une propriété uniquement pour le chemin dans le graphe de flot de contrôle pris lors de l'exécution de l'application. Les solutions dynamiques ne permettent pas de vérifier que certaines propriétés de sécurité sont vérifiées dans tout le programme. Les solutions statiques se basent sur des états du programme sans

aucune exécution préalable de celui-ci, ce qui permet d'explorer tous les chemins d'exécution de graphe de flot de contrôle. Il est ainsi possible de vérifier certaines propriétés de façon globale sur l'application, peu importe le chemin pris dans le graphe de flot de contrôle.

1.3 Mécanismes de contrôle de flux d'information statique

Utiliser un mécanisme de contrôle de flux d'information statique permet de garantir des propriétés de sécurité en analysant le code de l'application à contrôler. La vision du contrôle de flux d'information est cloisonnée à l'application et dépend fortement des informations fournies par l'auteur du programme. Les solutions au niveau des langages de programmation sont nombreuses et répertoriées par Sabelfeld et Myers [29].

Une analyse statique examine les chemins d'exécution possibles que le programme peut potentiellement prendre lors de son exécution, et utilise les informations à sa disposition dans le code de l'application pour vérifier que les propriétés de sécurité souhaitée sont vérifiées.

Cette technique a un avantage considérable : l'analyse se fait sans exécuter le programme, il n'y a donc pas de surcoût lors de son exécution. Elle se fait généralement lors de la compilation du programme, là où le code source de l'application est encore présent et plus aisément analysable grâce notamment aux informations de typage.

Comme toute analyse statique, il est nécessaire de faire des compromis entre une analyse correcte ou complète.

Correction : si l'analyse établit qu'une propriété est vérifiée dans le programme analysé, alors le programme vérifie bien cette propriété.

Complétude : si le programme analysé vérifie une propriété, alors l'analyse de ce programme confirmera que ce programme vérifie bien la propriété.

Avoir une analyse statique à la fois correcte et complète est malheureusement impossible, en vertu notamment du théorème de Ryce [30, 31]. En général, les analyses statiques sont correctes par construction. L'inconvénient majeur de ces approches réside alors dans la surapproximation qui mène parfois à de faux positifs (les approches rejettent des programmes corrects).

Plusieurs approches ont vu le jour, certaines approches se basent sur la sémantique du langage de programmation pour propager l'étiquette tout au long des chemins potentiels d'exécution, d'autres utilisent les systèmes de types qui ont fait leurs preuves depuis des décennies.

1.3.1 Approche basée sur la sémantique du langage

Une solution basée sur la sémantique des langages a permis de formaliser la sécurité des applications en termes de comportement du programme.

Denning et Denning [32] présente un mécanisme de certification pour vérifier les flux d'information dans un programme. Ce mécanisme de certification s'appuie sur un modèle de politique de sécurité sous forme de treillis et une relation qui représente les flux d'information autorisés entre les classes de sécurité. Ce mécanisme de certification, qui est historiquement une des premières approches proposées spécifiquement pour contrôler les flux d'information, peut être inclus dans les phases d'analyse de la plupart des compilateurs.

Les auteurs se sont concentrés sur la résolution du problème du confinement énoncé par Lampson [27] : comment prouver qu'un programme ne fait en aucun cas dépendre une donnée publique selon une donnée censée être confidentielle ?

Denning et Denning [32] considèrent que toute constante est membre du niveau de sécurité le plus bas $\bot$. Par exemple, l'affectation de la valeur « 42 » à la variable x est donc marquée $\bot$. Les flux implicites sont pris en charge, ainsi que la gestion des exceptions uniquement lorsque l'exception est définie explicitement par le programme.

L'inconvénient principal de cette approche est qu'il est nécessaire d'attacher un niveau de sécurité pour chaque conteneur d'information (variables, fichiers, etc.) de façon statique lors de leurs déclarations dans le code source du programme. L'évolution dans le temps du système des niveaux de sécurité des fichiers qui peuvent être modifiés n'est donc pas prise en compte. Cette solution permet une représentation sous forme de treillis des niveaux de sécurité, mais la politique de flux d'information doit être spécifiée avant la compilation du programme, elle est donc statique et immuable au cours de la vie du système. Cette certification fige donc la preuve au moment où elle a été réalisée et ne permet pas de garantir qu'un fichier initialement marqué comme public n'ait pas été modifié avec des données confidentielles par un autre programme ou utilisateur.

Ce type d'approche manque de flexibilité et il est fastidieux d'implémenter une telle analyse pour chaque langage de programmation populaire. Les chercheurs se sont donc penchés sur des solutions plus flexibles et génériques. Une approche assez courante dans la littérature consiste à implémenter une analyse en s'inspirant des systèmes de types. Cela permet plus de flexibilité, en autre grâce aux langages de programmation typés comme OCaml ou Java.

1.3.2 Approche basée sur les systèmes de typage statique

L'utilisation des systèmes de types pour le contrôle de flux d'information est une solution prometteuse et très populaire dans la littérature depuis les travaux précurseurs de Volpano et Smith [24]. Cette approche permet de garantir la propriété de non-interférence et peut être déployée pour les langages de programmation haut niveau [33, 34] et bas niveau [35, 5].

Les langages de programmation haut niveau ont plusieurs avantages qui facilitent l'analyse. Ils intègrent un système de types pour vérifier que les types déclarés dans le programme respectent la spécification du langage. L'ajout en sus du type ordinaire d'une métadonnée exprimant le niveau de sécurité pour chaque expression du programme est une solution simple et rapide. L'expressivité de ces langages permet de gérer des structures de données (classes, objets, tableaux, etc.) et des évènements comme les exceptions. Les langages de programmation bas niveau ne possèdent pas ces atouts qui rendent donc leur analyse beaucoup plus compliquée.

Dans cette partie, nous traiterons uniquement des systèmes de types statiques qui sont donc exécutés lors de la compilation du programme. Les systèmes de types dynamiques seront eux traités dans la section 1.4.1.

Les langages de programmation haut niveau basés sur le lambda calcul ont beaucoup été employés de par leur base formelle qui permet de prouver plus facilement l'exactitude du système de type mis au point pour le contrôle de flux d'information. Heintze et Riecke [23] introduisent par exemple des annotations qui permettent de faire du contrôle de flux d'information pour du lambda calcul.

Volpano, Irvine et Smith [36] ont proposé un système de type pour un langage impératif simple, qui garantit une propriété de non-interférence. Ils s'appuient sur les travaux préliminaires de Denning [26] et proposent un modèle de politique de flux d'information sous forme de treillis. La valeur ajoutée de cette solution est que l'analyse est prouvée correcte, ce qui n'était pas le cas des travaux précédents. Ainsi, tout programme correctement typé garantit la propriété de non-interférence : les variables d'un programme ayant un niveau de sécurité n n'interfèrent pas avec des variables de niveau de sécurité inférieur $< n$. Les flux d'information explicites et implicites sont pris en compte lors de l'analyse.

Ces résultats ont inspiré différents travaux qui se sont attachés à implémenter une approche similaire pour des langages de programmation plus complexes et couramment employés. Ainsi, Myers et Liskov [33] ont développé Jif, une extension du langage de programmation Java, qui permet d'analyser de façon statique les flux d'information. Le choix du langage Java

est très intéressant car il possède également un environnement d'exécution, ce qui a permis par la suite d'améliorer Jif en ajoutant des fonctionnalités nécessitant un aspect dynamique, nommé JFlow [37] que l'on traitera dans la section 1.4.1 de cet état de l'art.

Pottier et Simonet [34] ont implémenté une extension du langage OCaml en étendant le système de type pour vérifier automatiquement que les flux d'information dans le programme respectent une politique de flux d'information. De la même façon que les autres implémentations, il est nécessaire d'ajouter des annotations au système de type pour y attacher un niveau de sécurité défini par un treillis. Pottier et Conchon [38] proposent une solution permettant d'étendre n'importe quel système de type par une analyse de dépendance pour prendre en charge les flux d'information. Ils tirent également parti du système d'inférence de type de OCaml qui évite au programmeur de devoir spécifier le type de sécurité de toutes les variables du programme.

Barthe, Pichardie et Rezk [35] ont proposé un système de type pour le Bytecode Java, qui est un langage de plus bas niveau, interprété par une machine virtuelle Java. Ces travaux ont de plus été formalisés et prouvés corrects à l'aide de l'assistant de preuve Coq.

L'analyse des flux d'information pour les langages assembleur pose un challenge de taille à cause du fossé sémantique qui sépare les langages haut niveau et bas niveau. En effet, l'absence de structures de flux de contrôle et d'information sur les types dans les langages assembleur empêche une analyse précise des flux d'information. Medel, Compagnoni et Bonelli [5] ont ainsi abordé ce challenge en définissant un langage assembleur typé nommé SIF. Cette solution utilise deux instructions assembleur afin de prendre en compte les flux d'information implicites en empilant le niveau de sécurité du registre PC. Cela implique concrètement la modification du cœur du processeur afin d'y insérer la logique de ces deux instructions. Medel, Compagnoni et Bonelli [5] prouvent ainsi que tout programme écrit en SIF bien typé respecte la propriété de non-interférence.

Les mécanismes de contrôle de flux d'information statiques n'ont pas d'impact sur les performances à l'exécution, car l'analyse est réalisée durant la phase de compilation. Le programmeur doit néanmoins étiqueter les conteneurs d'information qu'il utilise dans le code source de l'application. Ces étiquettes sont statiques et immuables durant la vie du système. Ainsi si le code associe un niveau *public* à un fichier *notes.txt*, même si ce fichier *notes.txt* contient plus tard des données avec le niveau de sécurité *secret*, l'étiquette restera *public* ou devra être modifiée dans le code source et analysée de nouveau.

L'hypothèse commune dans plusieurs travaux est que la politique des flux

d'information est connue statiquement lors de la phase de compilation, ce qui est une hypothèse pas toujours réaliste, car les informations dans un système évoluent constamment. En outre, tout programme peut produire et recevoir des exceptions ce qui change le flux de contrôle du programme et inclut donc des flux d'information implicites. La prise en compte de ses exceptions est mal gérée statiquement car ce sont des évènements dynamiques qui se produisent lors de l'exécution de l'application. Ce manque de flexibilité des solutions statiques a poussé les chercheurs à se pencher sur des solutions dynamiques, plus flexibles.

1.4 Mécanismes de contrôle de flux d'information dynamique

Vérifier les flux d'information en même temps que l'exécution du programme implique un surcoût en termes de performance, mais permet d'offrir plus de souplesse par rapport aux politiques de sécurité. L'aspect dynamique permet également de diminuer la surapproximation de l'analyse. On parle alors de mécanisme de DIFT.

L'aspect dynamique du contrôle des flux d'information peut être implémenté dans l'environnement d'exécution de l'application, directement dans l'application grâce à une instrumentation ou directement dans le système d'exploitation.

1.4.1 Approche basée sur l'environnement d'exécution

Un environnement d'exécution est une couche logicielle qui permet soit d'émuler à la volée une architecture matérielle (par exemple QEMU [39]), soit d'interpréter un langage de programmation (par exemple Perl [25]), soit d'exécuter un programme grâce à la compilation à la volée appelée Just-In-Time (JIT) (par exemple Javascript [40]).

Java est un langage de programmation avec un environnement d'exécution riche en fonctionnalités de sécurité, par exemple la vérification du bytecode. Il a donc naturellement été choisi par beaucoup d'approches pour y ajouter un système de contrôle de flux d'information dynamique grâce à son implémentation au niveau de la machine virtuelle Java.

Myers et Myers [37] propose JFlow, une extension pour le langage de programmation Java, qui outre la vérification statique des annotations dans le code source, propose une vérification des étiquettes lors de l'exécution. Basé sur Jif [33], JFlow permet d'ajouter un aspect dynamique au mécanisme de flux d'information. Dans JFlow, les étiquettes peuvent également

être utilisées comme des valeurs de première classe, ainsi il est possible d'utiliser une étiquette comme paramètre d'une fonction. Cette fonctionnalité permet de gérer les cas dans lesquels une étiquette ne peut pas être inférée de façon statique par le système de type. Toutefois, cela nécessite que le programmeur prenne en compte cette problématique dans le développement du logiciel. JFlow fournit un modèle de contrôle de flux d'information décentralisé développé par Myers, Myers et Liskov [41], qui permet ainsi à chaque utilisateur d'appliquer la politique de sécurité des données qui lui appartient. Ainsi il n'y a pas d'autorité de confiance qui fixe la politique pour tous les utilisateurs. Myers, Myers et Liskov [41] énoncent que JFlow satisfait la propriété de non-interférence bien qu'ils ne le prouvent pas formellement.

Perl propose un système de propagation de teintes qui offre la possibilité d'initialiser, de propager et de vérifier les niveaux de sécurité des variables d'un programme [25]. Perl permet donc de faire du contrôle de flux d'information à grain moyen. Seuls deux niveaux de sécurité sont proposés, un niveau pour représenter les données authentiques et un niveau pour représenter les données souillées. Une partie de la politique de sécurité est également figée puisque toutes données provenant de fichiers ou de certains appels système sont d'office marquées comme souillés.

Navaki Arefi et al. [8] introduisent FAROS, un mécanisme de flux d'information dynamique pour le système d'exploitation Windows s'appuyant sur le logiciel de machine virtuelle QEMU et son plugin d'analyse dynamique PANDA. FAROS permet de faire du suivi de flux d'information à grain fin et permet donc de marquer chaque octet dans la mémoire. De plus, il propose un suivi des flux d'information dans toutes les couches du système. Il est donc possible de connaître la provenance exacte d'un octet dans la mémoire. La couche matérielle est également tracée, mais se traduit par du logiciel puisque la solution se base sur une machine virtuelle. FAROS gère les flux d'information implicites. L'inconvénient principal de cette solution est qu'elle est purement logicielle, et donc introduit un surcoût important. Lorsqu'un fichier est chargé en mémoire, son étiquette est propagée vers une zone mémoire utilisée par celui-ci. Pour pouvoir faire cette propagation, un pilote a été développé et des fonctions spécifiques de propagation des labels sont appelées lors d'un appel système. Lorsqu'un bloc de base est exécuté dans le système invité (Windows), le plugin développé dans QEMU récupère les instructions CPU contenues dans le bloc de base et propage les étiquettes. FAROS présente de bons résultats concernant la détection d'attaques Return Oriented Programming (ROP). Les auteurs expliquent que leur priorité n'est pas la performance or c'est une caractéristique importante pour qu'une solution de DIFT puisse être utilisée.

1.4.2 Approche basée sur l'instrumentation de code binaire

L'instrumentation de code binaire est une technique permettant d'ajouter des instructions à un programme. Cette technique peut être utilisée avant ou durant l'exécution du programme, à la volée (Dynamic Binary Instrumentation (DBI)).

Analyser du code machine est difficile car le code binaire manque de structure, de types et utilise parfois des adresses symboliques qui sont résolues juste en amont ou durant l'exécution du programme par l'éditeur de lien (*relocation*). Il existe principalement deux méthodes d'instrumentation :

L'utilisation d'un trampoline, qui introduit un flux de contrôle additionnel comme une instruction de saut vers une fonction contenant la charge utile à exécuter. Cette charge utile peut par exemple être des instructions qui permettent d'envoyer des informations vers un moniteur externe. C'est cette méthode qui est utilisée pour résoudre dynamiquement des adresses par l'éditeur de liens grâce à la Procedure Linkage Table (PLT).

L'utilisation d'un patch, qui permet d'insérer directement les instructions de la charge utile à un endroit précis dans le code de l'application. L'injection de nouvelles instructions dans le code de l'application nécessite de décaler les instructions suivantes et de recalculer les adresses pointant vers le code de l'application impacté par le décalage.

Kim et al. [42] proposent RevARM, un outil permettant la réécriture de fichiers exécutables basés sur les architectures ARM à l'aide de l'instrumentation. Les auteurs considèrent qu'il est important de prendre en considération le fait que les applications ne sont pas fournies avec leur code source. Il est donc important d'utiliser des techniques qui permettent à partir d'un fichier exécutable compilé, de le réécrire dans le but d'ajouter des fonctionnalités de sécurité. RevARM utilise l'instrumentation basée sur l'utilisation de patchs qui permettent d'insérer directement les nouvelles instructions au point cible voulu sans créer de trampoline ou de nouveaux flux de contrôle. Cela permet de ne pas introduire de surcoût dû aux instructions de saut et de branchements liés au trampoline. RevARM gère plusieurs formats de fichiers populaires, par exemple le format Executable and Linkable Format (ELF), et procède à l'instrumentation avant l'exécution du programme, car l'instrumentation à la volée génère un surcoût en termes de temps d'exécution et d'espace mémoire qui constitue un inconvénient majeur dans les systèmes embarqués. RevARM n'est qu'une solution d'instrumentation ciblant les applications liées à la sécurité et n'inclut donc pas de contrôleur de

flux d'information. Malgré tout, nous utiliserons une approche similaire pour l'instrumentation qui présente un faible surcoût pour les systèmes embarqués utilisant une architecture ARM.

Les solutions de contrôle de flux d'information purement logicielles sont principalement basées sur l'instrumentation dynamique de binaire DBI, ce qui ralentit le programme utilisateur de 5 à 100 fois. Les outils permettant d'instrumenter dynamiquement un programme tel que Dynamo [43], DynamoRIO [44], Pin [45], Valgrind [46], etc. sont souvent employés dans les solutions de DIFT.

Newsome et Song [6] proposent une solution de DIFT se basant sur l'instrumentation de fichiers exécutables à la volée (DBI) appelée TaintCheck. Les auteurs expliquent que leur solution permet de détecter l'exploitation de différents types de vulnérabilités. Il n'y a que deux niveaux de sécurité, un premier niveau pour marquer les données provenant de sources de confiances et un second niveau pour identifier les données provenant de sources suspicieuses. Newsome et Song [6] n'expliquent pourtant pas comment est faite l'affectation des niveaux de sécurité pour marquer les sources. Les valeurs littérales sont considérées d'office de confiance, ce qui ne permet donc pas de garantir une propriété de non-interférence car les flux implicites ne sont pas tous pris en compte. TaintCheck ralentit l'exécution de l'application entre 1,5 et 40 fois.

Qin et al. [7] proposent LIFT, une solution exclusivement logicielle à l'aide de StarDBT, un outil d'instrumentation à la volée pour le code binaire des applications. Développée pour Windows, leur approche permet de détecter plusieurs types d'attaques. Les auteurs de cet article ne s'intéressent pas uniquement aux fuites d'information, mais également à détecter des attaques comme la modification d'adresse de retour (ROP). Le temps d'exécution des applications utilisant LIFT est tout de même en moyenne de 3,6 à 6,2 fois plus que le temps d'exécution normal.

Un des avantages de LIFT est qu'il permet également de faire du suivi de flux d'information pour le code contenu dans les bibliothèques externes, et donc de couvrir la totalité du code. LIFT n'utilise qu'un seul bit pour représenter les niveaux de sécurité. Ces deux niveaux de sécurité représentent uniquement le niveau de confiance du canal par lequel est arrivée la donnée. De plus, seuls les flux d'information explicites sont traités, les flux implicites sont totalement ignorés. LIFT prend en compte la sémantique de certaines opérations spéciales qui sont utilisées pour réinitialiser les valeurs d'un registre CPU comme illustré dans le code 1.3. Lors de l'exécution de ces instructions de réinitialisation, il est également important de réinitialiser le tag correspondant du registre concerné. Le code 1.3 présente un exemple d'une telle instruction.

```
1 mov r1, #0
```

CODE SOURCE 1.3 – Exemple de réinitialisation du registre r1

Lors de l'exécution de l'instruction présente dans le code 1.3, LIFT doit également réinitialiser le niveau de sécurité du registre concerné, car sa valeur précédente est maintenant incorrecte. Dans ce cas, l'affectation du niveau de sécurité de $r1$ par le plus bas niveau de sécurité est une possibilité : $\underline{r1} = \bot$.

Les solutions basées uniquement sur l'instrumentation ont un impact important en terme de temps d'exécution. Elles ne proposent généralement pas plus de deux niveaux de sécurité, ce qui les restreint à émettre de fortes hypothèses de départ sur les sources à considérer de confiance ou non. De plus, le manque de communication avec le système d'exploitation ne permet pas de propager les niveaux de sécurité vers les fichiers du système.

1.4.3 Le contrôle de flux d'information dans les systèmes d'exploitation

Le développement d'un support pour le DIFT dans les systèmes d'exploitation est très complexe à mettre en oeuvre car des flux d'information se produisent de façon concurrente entre différentes structures de données et plusieurs processus. Deux approches ont donc été développées : repartir de zéro en créant un système d'exploitation intégrant directement le support du DIFT [47, 2], ou développer un support DIFT pour un système d'exploitation déjà existant [48, 49, 50].

L'agence de sécurité des États-Unis d'Amérique National Security Agency (NSA) avait besoin d'un système d'un système d'exploitation capable de gérer des politiques de sécurité basées sur des politiques de sécurité multiniveau comme celles proposées par DENNING [26] et BELL et PADULA [13]. La NSA a donc développé SELinux et a proposé à la communauté Linux d'adopter les modifications dans le noyau Linux, mais le souhait d'une solution générique et modulable a poussé la communauté Linux à développer un système de modules de sécurité appelée Linux Security Modules (LSM) [51]. De nombreuses solutions DIFT [48, 52, 50, 49] ont tiré profit des LSM pour développer leurs solutions.

Le développement d'une solution DIFT dans le système d'exploitation, illustré par la Figure 1.7, est un choix raisonnable car par définition, le système d'exploitation est une zone protégée par différents mécanismes de sécurité. En particulier, le noyau du système d'exploitation s'exécute générale-

ment dans un mode privilégié du CPU, ce qui lui permet d'isoler son espace mémoire. De plus, le système d'exploitation est le garant des ressources du système pour les applications utilisateur, il a donc une vision globale des structures de données utilisées pour chaque processus utilisateur. Il a également un accès au système de fichier et peut vérifier et contrôler tous les appels système effectués par les applications utilisateur.

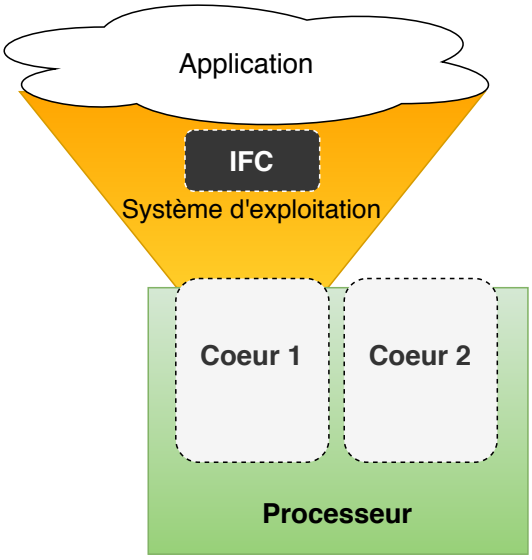


FIGURE 1.7 – Suivi de flux d'information au niveau du système d'exploitation.

EFSTATHOPOULOS et al. [47] ont développé Asbestos, un système d'exploitation qui permet de faire du contrôle de flux d'information et de l'isolation de services. Asbestos supporte des étiquettes multiniveau sous forme de treillis, ce qui donne lieu à des politiques de sécurité complexes. La gestion des étiquettes tire profit des modèles de contrôle d'accès MAC et DAC. En effet, Asbestos permet à la fois d'appliquer une politique de flux d'information globale, et à chaque processus de créer et de gérer dynamiquement sa propre politique de flux d'information. Les processus dans Asbestos possèdent deux étiquettes, la première représente le marquage actuel du processus tandis que la deuxième représente le niveau de sécurité maximum que le processus peut accepter provenant des autres processus. La communication entre les processus se fait par passage de messages qui sont considérés comme un conteneur d'information à part entière et joue un rôle dans la propagation et la contrôle des flux d'information.

ZELDOVICH [2] propose HiStar, un nouveau système d'exploitation développé à partir de zéro avec comme objectif principal d'intégrer un système de

contrôle de flux d'information dans sa conception. HiStar emprunte la même notion de marquage des processus que celle d'Asbestos.

HiStar et Asbestos ont démontré qu'il était possible d'implémenter avec succès le DIFT au niveau du système d'exploitation. Toutefois, il s'agit dans les deux cas d'OS réécrits entièrement dans le but d'implémenter le DIFT. Ces OS utilisent en outre des Application Programming Interface (API) spécifiques et ne sont pas compatibles avec les OS existants, utilisés par la plupart des applications, comme Linux, Mac OS ou Windows. Cela limite leur utilisation, y compris dans un cadre de recherche, car il est nécessaire de redévelopper les applications pour ces OS, ce qui constitue un effort d'ingénierie conséquent. En outre, maintenir un OS sur le long terme est une tâche fastidieuse et coûteuse. Des travaux se sont donc intéressés à implémenter le DIFT sur des OS existants, notamment Linux.

Viet Triem Tong, Clark et Mé [53] proposent une approche de contrôle d'accès dynamique basée sur la modélisation des flux d'information entre les conteneurs d'information. Ainsi, la décision d'accorder ou non l'accès d'un utilisateur à un fichier n'est pas statique mais dépend des informations actuellement contenues dans celui-ci. Ceci permet de faire du contrôle de flux d'informations à grain moyen, car les conteneurs d'information sont les structures de données utilisées par le système d'exploitation comme les fichiers, les zones mémoire des processus, etc. La politique de flux d'information est décentralisée, de cette manière chaque propriétaire de fichier peut spécifier explicitement quel mélange d'information est autorisé et qui peut y accéder. Comme dans Asbestos, chaque conteneur d'information possède une collection d'étiquettes décrivant le niveau de sécurité actuel, appelé Information tag (itag), et une collection d'étiquettes décrivant la politique de flux d'information Policy tag (ptag). Les auteurs proposent une implémentation de cette approche sous Linux, appelée Blare, qui utilise les modules de sécurité de Linux (LSM)

Toutefois, dans ces travaux, aucune vérification n'est faite pour vérifier que toute action générant un flux d'information est bien capturée par les crochets LSM. Georget et al. [1] ont identifié certains cas dans lesquels les flux d'information échappaient au système à cause de crochets LSM mal placés ou manquants et de certains problèmes de concurrence. Georget et al. [1] ont donc corrigé ces problèmes et ont formellement prouvé avec Coq que tous les points d'ancrage LSM présents permettent de capturer tous les flux d'information qui se produisent. Cette amélioration a donné naissance à RfBlare qui permet de suivre les flux d'information dans un système Linux 4.7, de façon prouvée, complète et robuste aux situations de concurrence dans le système.

Les solutions de DIFT dans les systèmes d'exploitation sont des solutions

avec une granularité à gros grains, elles peuvent arbitrer les accès aux objets et ressource du système mais ne permettent pas de propager les niveaux de sécurité vers des objets à grain fin comme les structures de données des applications utilisateur. Cela conduit à des surapproximations importantes, car les flux d'informations sont seulement inférés à partir de l'observation des appels système. Dès lors qu'un programme accède en lecture à un conteneur A (typiquement un fichier ou une page mémoire) puis qu'il écrit dans un second conteneur B, les solutions de DIFT au niveau OS considèrent qu'un flux d'information est généré de A vers B. Toutefois, cela n'est pas nécessairement le cas, suivant les opérations réalisées par l'exécution des instructions du programme en question.

Roy et al. [48] ont essayé de résoudre ce problème en proposant Laminar, une solution combinant un système d'exploitation Linux modifié pour le DIFT à l'aide des crochets LSM, et une machine virtuelle Java supportant le suivi de flux d'information à grain fin dans les applications Java. La partie Java de Laminar ne permet pas de faire du suivi de flux d'information dans toutes les données de l'application, seuls les objets alloués dynamiquement sont aptes à être marqués et les applications doivent être développées avec du code propre à Laminar. En effet, il revient au programmeur de l'application d'expliciter certains flux d'information et de faire parfois appel à des fonctions de manipulation de tags directement dans le code de l'application.

Les solutions de DIFT au niveau OS permettent de surveiller l'ensemble des applications d'un système d'exploitation, avec un impact limité sur les performances à l'exécution. En outre, les approches qui s'appuient sur des systèmes d'exploitation existants ne nécessitent pas de recompiler les applications. Toutefois, toutes ces approches souffrent d'un problème de surapproximation des flux qui peut conduire à des faux positifs. Une solution permettant de remédier au manque de précision, tout en permettant de suivre les flux entre les applications et les différents conteneurs du système, consiste à implémenter le DIFT à différents niveaux du système : un suivi à gros grain est réalisé au niveau de l'OS et, pour les programmes complexes qui le nécessitent, le suivi des flux internes de l'application est réalisé par un DIFT à grain fin, au niveau du programme. Laminar, qui implémente cette approche, présente cependant des limitations. Le suivi à grain fin, implémenté au niveau de la Java Virtual Machine (JVM), ne s'applique qu'aux programmes Java. En outre, ce suivi nécessite un effort de la part du programmeur. Afin de suivre un nombre important de programmes, développés dans différents langages de programmation, il est nécessaire de combiner une approche de DIFT au niveau OS avec une approche implémentant le DIFC au niveau du binaire. Dans ce contexte, une des solutions pour optimiser le suivi de flux d'information à grain fin consiste à utiliser une solution matérielle capable

de prendre en charge le suivi des flux d'information qui se produisent entre les registres CPU et la mémoire du système.

1.4.4 Le contrôle de flux d'information réalisé matériellement

Les problèmes de performance en temps d'exécution des solutions logicielles ont fait pencher les chercheurs vers l'étude de solutions matérielles. Trois principales approches ont alors émergées :

— *In-core* : le DIFT est implémenté par des circuits logiques dédiés dans les différents étages du *pipeline* de chaque cœur du CPU ;
— *Off-loading* : le DIFT est implémenté par un programme qui s'exécute sur un des cœurs du CPU, qui est dédié au DIFT ;
— *Off-core* : le DIFT est implémenté sur un coprocesseur dédié.

1.4.4.1 In-core

Cette approche, illustrée par la Figure 1.8, est intrusive puisqu'elle consiste à modifier le cœur du CPU pour y implémenter toute la logique de propagation, de calcul et de stockage des étiquettes à chaque étage du pipeline.

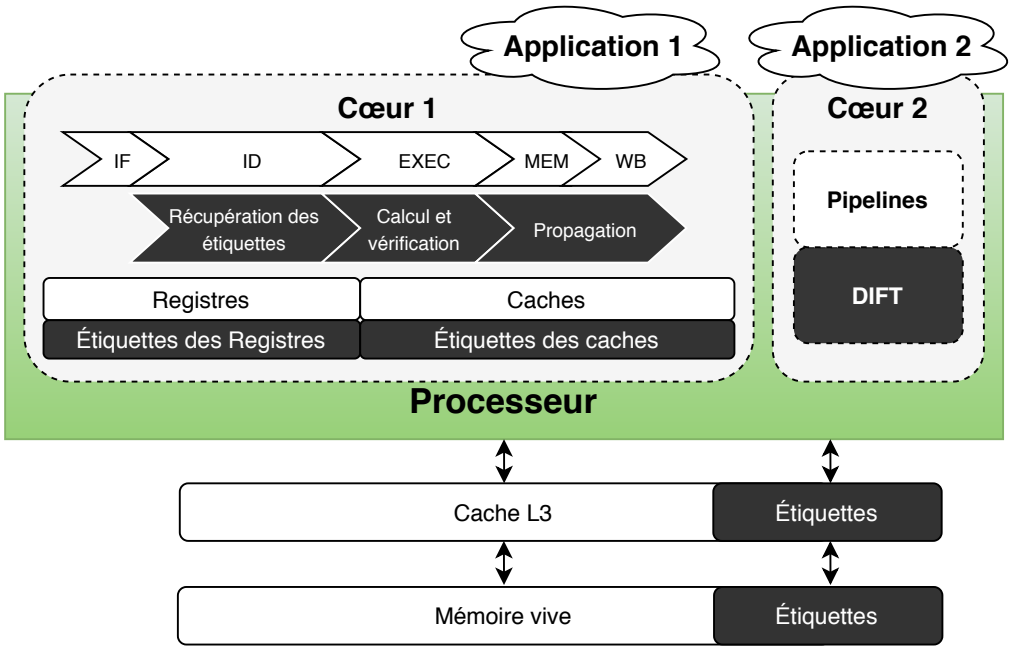


FIGURE 1.8 – Solution In-core.

Suh et al. [3] proposent une approche dans laquelle le système d'exploitation identifie les canaux d'entrée suspects et le processeur suit les flux d'information depuis ces canaux. Grâce à ce suivi, le processeur peut par exemple vérifier que l'adresse cible d'une instruction de saut provient d'un canal suspect ou non. Si c'est le cas, la politique de sécurité est alors de générer une exception puis de laisser le système d'exploitation traiter l'exception. La politique de sécurité est spécifiée par le système d'exploitation et est ensuite vérifiée par le processeur. Suh et al. [3] utilisent un bit pour pouvoir représenter le niveau de sécurité. Il y a donc seulement deux niveaux de sécurité, *authentique* et *suspect*. Les flux implicites sont pris en charge, grâce à une analyse statique vers l'arrière pour retracer la chaîne de dépendance des instructions qui influencent la condition du branchement. Dalton, Kannan et Kozyrakis [21] présentent Raksha, une solution DIFT qui propose cette fois une représentation des étiquettes de 4 bits par mot machine. Ce choix est guidé par le fait que Raksha permet l'exécution de 4 politiques de sécurité en parallèle. Tous les registres CPU, la mémoire vive et le cache de données sont donc étiquetés.

Les solutions In-core ont plusieurs limitations. Tout d'abord, cette approche est difficile à industrialiser car les grandes compagnies de conception de processeur n'ont pas comme priorité de proposer du suivi de flux d'information. De plus, le support du DIFT implique une complexification de la conception des CPU qui rend plus difficiles les étapes de validation. En pratique, l'évaluation de ces approches est souvent limitée à la simulation ou via des Softcore, c'est à dire des CPU implémentés dans un système reprogrammable comme un Field Programmable Gate Array (FPGA). Cette approche est également compliquée dans un contexte multicœur : en effet, il est nécessaire de faire communiquer les systèmes de DIFT de chaque cœur du CPU (par exemple, si l'application utilisateur est commutée du cœur 1 vers le cœur 2). Enfin, cela induit des surcoûts en termes de consommation énergétique qui sont non négligeables pour des CPU, notamment ceux destinés à des systèmes embarqués.

1.4.4.2 Off-loading

Cette approche illustrée par la Figure 1.9, réserve un deuxième cœur CPU pour le dédier au contrôle de flux d'information. Contrairement à l'approche précédente, elle ne nécessite pas de modifier en profondeur la microarchitecture des CPU. Elle est donc plus susceptible d'être adoptée et elle peut être implémentée sur des processeurs existants.

En raison des protections matérielles développées par les concepteurs de processeur pour isoler strictement les cœurs entre eux, le cœur dédié au DIFT

(numéro 2 dans l'exemple) ne peut pas connaître l'état des autres cœurs, qui exécutent les programmes utilisateurs (cœur 1 dans l'exemple). Ce fossé sémantique ne lui permet pas de propager les étiquettes selon l'état des applications. Une communication intercœur et interprocesseur est donc nécessaire afin de faire transiter les informations nécessaires à la propagation entre les cœurs exécutant les applications et le cœur dédié au DIFT (soit du cœur 1 au cœur 2 dans l'example).

Cette approche a été utilisée par CHEN et al. [9] , TOWNLEY et al. [54] et RUWASE et al. [55] dans leurs solutions. Toutes ces solutions se basent sur une architecture appelée Log-Based Architectures (LBA) développée par CHEN et al. [56], qui repose sur l'extraction vers un deuxième cœur d'un journal d'évènements du cœur exécutant l'application utilisateur. Cette solution est

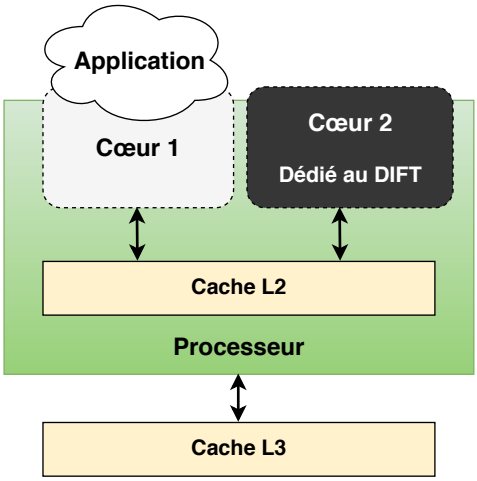


FIGURE 1.9 – Solution d'Off-loading.

tout de même très coûteuse puisque les caches L2 et L3 sont principalement utilisés pour la transmission du journal des évènements. JEE et al. [57] choisissent un autre moyen de communication entre cœurs exécutant l'application et le cœur du processeur dédié au DIFT : l'instrumentation dynamique de binaires dans une machine virtuelle à l'aide de Pin [45].

Le principal problème de cette approche est qu'elle nécessite d'avoir un deuxième coeur généraliste pour réaliser le DIFT : dans un contexte où le système est embarqué et donc soumis à des contraintes de consommation, cette solution est difficilement acceptable.

1.4.4.3 Off-core

Cette approche, illustrée par la Figure 1.10, est basée sur l'utilisation d'un coprocesseur dédié au DIFT.

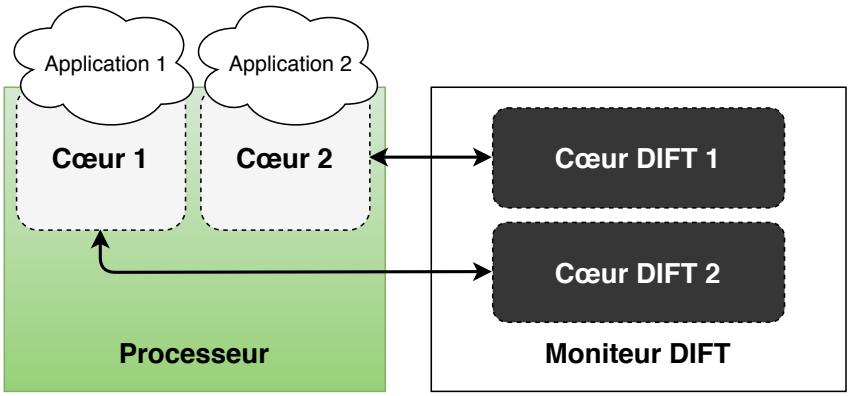


FIGURE 1.10 – Solution Off-core.

KANNAN, DALTON et KOZYRAKIS [4] présente une solution proposant un coprocesseur pour le DIFT sous forme de Softcore que l'on nommera Raksha v2. Ce prototype est basé sur un processeur principal LEON3 (architecture SPARC V8) fonctionnant sous Linux et un coprocesseur inspiré de Flexi-Taint [58], qui utilise l'approche introduite par l'architecture DIVA [59] et qui permet d'effectuer toutes les opérations nécessaires au DIFT dans les derniers étages du coprocesseur. L'échange d'informations entre le processeur et le coprocesseur se produit en utilisant le cache partagé L2 et L3. KANNAN, DALTON et KOZYRAKIS [4] ont modifié le noyau Linux pour avoir un support pour le DIFT mais ne mentionnent pas les problématiques liées aux flux d'informations dans le noyau. Les communications interprocessus ne sont pas prises en charge, il est donc impossible de gérer correctement les flux d'information qui se produisent entre plusieurs fils d'exécution ou processus. En cas de violation de la politique de sécurité, une exception est levée et le contrôle est transféré aux moniteurs de sécurité. La synchronisation des opérations DIFT se produit uniquement lors des appels système. Ainsi, c'est lors des appels système que le CPU se met en pause le temps que le coprocesseur termine de propager les étiquettes. Raksha permet de représenter le niveau de sécurité d'un conteneur d'information grâce à 4 bits. Comme dans la première version de Raksha [21], le coprocesseur supporte quatre politiques de sécurité en parallèle.

Le principal goulot d'étranglement des solutions Off-loading et Off-core est la communication entre le cœur principal exécutant l'application utilisa-

teur à tracer et le coprocesseur chargé du DIFT. En effet, il est nécessaire d'extraire des traces sur l'état de l'application à analyser.

Des travaux récents proposent de tirer profit des composants de débogage et de traces fournis par les concepteurs de processeur. Cette approche permet de grandement faciliter l'extraction des informations sur l'état de l'application exécutée. Jindal et al. [60] proposent DHOOM, une solution comprenant un moniteur de sécurité Off-Core développé sur FPGA pour la vérification d'assertions lors de l'exécution d'un programme. Ce moniteur de sécurité reçoit des traces grâce à l'implémentation de composant de débogage et à la modification du processeur LEON3. Decker et al. [61] proposent RETOM, une approche Off-core utilisant les composants de débogage ARM Embedded Trace Macrocell (ETM) permettant ainsi de créer un moniteur externe au système sur puce avec aucun surcoût lors de l'exécution du programme concernant l'extraction des traces depuis le processeur vers un moniteur implémenté sur FPGA. Bien que DHOOM et RETOM ne sont pas explicitement utilisés pour faire du contrôle de flux d'information, ces approches permettent tout de même d'éliminer une partie du surcoût de la communication entre le processeur et le moniteur de sécurité.

1.5 Combinaison conjointe d'analyse statique et dynamique

Les moniteurs de flux d'information hybrides utilisent une combinaison d'analyses statiques et dynamiques pour garantir certaines propriétés de sécurité du système. Les moniteurs hybrides essayent donc de tirer profit des analyses statiques pour permettre d'identifier en amont les flux d'information qui se produisent dans une application sans introduire de coût supplémentaire lors de l'exécution, et l'analyse dynamique qui permet une plus grande flexibilité des politiques de sécurité et une plus grande précision.

Moore et Chong [62] montrent comment une analyse statique peut être utilisée pour améliorer les performances d'un moniteur de flux d'informations hybrides. Pour cela, ils proposent une analyse statique qui permet de déterminer lorsqu'il est intéressant de ne plus suivre dynamiquement les flux d'information d'une variable dans un programme, et permet donc de réduire le surcoût lors de l'exécution. Moore et Chong [62] se basent sur le moniteur de flux d'information de Russo et Sabelfeld [63], et seulement deux niveaux de sécurité sont possibles pour marquer les sources de confiance et les sources potentiellement malveillantes.

Dalton, Kannan et Kozyrakis [21] proposent une architecture pour

le suivi des flux d'information, grâce à une approche logicielle et matérielle. Ils proposent le support de politiques de sécurité flexible et programmable qui permet ainsi de se prémunir d'attaques de haut niveau comme des injections SQL, et de bas niveau comme des dépassements de tampon. Il est également possible de mettre en place plusieurs politiques de sécurité en même temps lors de l'exécution. Le système résultant est un poste de travail Linux complet pouvant appliquer des politiques de sécurité dans toutes les régions de la mémoire (tas, pile). Les architectures DIFT étendent chaque registre CPU et adresse mémoire par une étiquette de seulement un bit, deux niveaux de sécurité sont donc possibles.

1.6 Conclusion

Le contrôle de flux d'information est une technique prometteuse pour garantir des propriétés de confidentialité et d'intégrité d'un système. Le tableau 1.3 fait un état des lieux des contraintes et des fonctionnalités fournis par les différentes solutions de l'état de l'art.

Les solutions statiques sont basées sur la sémantique des langages de programmation. Bien quelles aient l'avantage de proposer une analyse en amont, elles ne permettent pas traiter de manière précise les flux d'information qui résultent d'évènements externes au programme et qui dépendent du contexte d'exécution comme les exceptions, les interruptions ou les appels systèmes. Pour y remédier, les auteurs proposent donc de figer la politique de sécurité en amont et de considérer un niveau de sécurité par défaut pour ces évènements externes (notamment les appels système).

On peut remarquer que les solutions [32, 36, 34, 25] basées sur les langages de programmation haut niveau sont par nature limitées par la granularité du suivi puisque ces solutions haut niveau n'ont aucune notion de l'architecture matérielle. Sabelfeld et Myers [29] font un état des lieux des techniques utilisées pour pouvoir garantir certaines propriétés de sécurité grâce aux flux d'informations, et donne quelques orientations de recherche. L'une d'entre elles est d'adapter les techniques d'analyse statique au langage assembleur afin de garantir des politiques de sécurité à grain fin au niveau de l'architecture matérielle et non plus au niveau du langage de programmation. Medel, Compagnoni et Bonelli [5] ont voulu explorer cette voie en essayant d'appliquer cette approche pour les langages bas niveau. Néanmoins, cette solution conduit à la création de nouvelles instructions et à la modification du cœur du processeur.

Les solutions dynamiques [6, 7] présentent quant à elle une bonne alternative puisque les évènements dynamiques du système lors de l'exécution du

programme sont pris en compte. Malheureusement les solutions dynamiques nécessitent des moyens de communication entre le moniteur de sécurité et le processeur exécutant le programme utilisateur qui deviennent alors rapidement un goulot d'étranglement. Dans la littérature, la représentation des politiques de sécurité dans les approches à grain fin et grain moyen utilise majoritairement deux niveaux de sécurité. Ainsi ces applications permettent seulement de marquer le niveau de confiance d'une donnée ou d'une source et ne permettent pas d'exprimer des politiques de sécurité complexes relative aux habilitations des utilisateurs. D'autre part, certaines approches nécessitent la modification interne de l'architecture du processeur [6], qui augmente donc la complexité du design et de la vérification de l'architecture. Cette forte contrainte décourage les entreprises de semi-conducteur à intégrer ces solutions de sécurité dans leurs architectures.

Les solutions de DIFT dans les systèmes d'exploitation [47, 1] offrent la possibilité d'intercepter les flux d'information qui surviennent dans les différentes structures de données du processus et de pouvoir suivre chaque évènement système. Elles manquent cependant de précision puisque les moniteurs au niveau OS n'ont aucune vision des flux d'information à grain fin qui se produisent lors de l'exécution d'une application utilisateur. Ce manque de précision oblige donc ces solutions à marquer par exemple toute une page mémoire de plusieurs kilo-octets, sans pouvoir y dissocier une variable secrète et une variable publique contenue dans cette page mémoire. Dans le but d'avoir une granularité fine, une approche proposée dans les travaux [8, 48] est de coupler le support du DIFT dans le système d'exploitation avec une machine virtuelle modifiée. Cette machine virtuelle peut ainsi récupérer les informations nécessaires pour faire du suivi de flux d'information dans la couche matérielle simulée par du logiciel. Dans un contexte multithread et multitâche, il est important que le système d'exploitation puisse capturer les flux d'information qui se produisent dans la mémoire partagée, les moyens de communication interprocessus, etc. Les systèmes d'exploitation ont tout de même l'avantage de pouvoir proposer des politiques de sécurité plus flexibles et complexes , mais également de pouvoir utiliser le système de fichier pour y sauvegarder l'état des flux d'information dans le système de façon persistante.

Les solutions matérielles reposent souvent sur la modification profonde de la couche matérielle afin d'y implémenter la logique de propagation des étiquettes ou pour extraire des informations sur l'état d'exécution du processeur. En outre, les politiques de sécurité sont souvent directement fixées dans le matériel.

L'introduction de nouveaux systèmes de débogage et de traces offerts par les concepteurs de processeur est également une bonne chose, puisqu'ils per-

mettent d'extraire des informations sur l'état d'exécution d'une application en temps réel, et sans surcoût en terme de temps d'exécution.

Solution	Approche	Aucune adaptation du code source de l'application	Liberté de choix du langage de programmation	Processeur original	Interface de communication	Granularité	Portée du contrôle	Représentation des étiquettes	Flexibilité des politiques de sécurité	Support d'exécution
Certification Denning et Denning [32]	Analyse statique (Sémantique du langage)	✗	✗	✓	✗	Grain moyen	Application	2 niveaux	✗	Non spécifié
Volpano Volpano, Irvine et Smith [36]	Analyse statique (Système de types)	✗	✗	✓	✗	Grain moyen	Application	2 niveaux	✗	Non spécifié
PottierIFML Pottier et Simonet [34]	Analyse statique (Système de types)	✗	✗	✗	✗	Grain moyen	Application	multiniveau	✗	Non spécifié
Perl *Perl Security* [25]	Analyse dynamique (Interpréteur)	✗	✗	✓	✗	Grain moyen	Application	2 niveaux	✗	Interpréteur Perl
SIF Medel, Compagnoni et Bonelli [5]	Analyse statique (Système de types)	✗	✗	✗	✗	Grain fin	Application	2 niveaux	✗	Non spécifié
TaintCheck Newsome et Song [6]	Analyse dynamique DBI	✓	✓	✗	✓ Instrumentation	Grain fin	Application	2 niveaux	✗	OS non spécifié
LIFT Qin et al. [7]	Analyse dynamique (DBI)	✓	✓	✓	✓ Instrumentation	Grain fin	Application	2 niveaux	✗	Windows
Asbestos Efstathopoulos et al. [47]	Analyse dynamique (OS)	✓	✓	✓	✗	Grain moyen	Structure de données du processus + OS	multiniveau	✓	Asbestos
RfBlare Georget et al. [1]	Analyse dynamique (OS)	✓	✓	✓	✗	Grain moyen	Structure de données du processus + OS	mult-niveau (32 bits)	✓	Distribution Linux 4.9
FAROS Navaki Arefi et al. [8]	Analyse dynamique Virtual Machine (VM)	✗	✓	✗	✓ QEMU	Grain fin	Application + OS	multiniveau	✓	Windows + QEMU
Laminar Roy et al. [48]	Analyse dynamique (VM + OS)	✗	✗	✓	✗	Grain moyen	Structure de données du processus + Application	multiniveau	✓	Linux + VM Java
Suh04 Suh et al. [3]	In-Core	✓	✓	✗	✗	Grain fin	Application	2 niveaux	✗	OS non spécifié
AddrCheck, MemCheck, TaintCheck and LockSet Chen et al. [9]	Off-Loading	✗	✓	✗	✓ Cache L2	Grain fin	Application	mult-niveau (1 à 64 bits)	✓	OS non spécifié
Raksha v2 Kannan, Dalton et Kozyrakis [4]	Off-Core	✗	✓	✗	✓ Propriétaire	Grain fin	Application	2 niveaux pour chaque politique (4 bits)	✗	Linux
HardBlare	Hybride (Off-Core + OS + Instrumentation)	✓	✓	✓ ARM Cortex-A9	✓ EMIO/AXI	Grain fin	Application + Structure de données du processus + OS	multiniveaux (32 bits)	✓	Distribution Linux 4.9

Table 1.3 – Tableau comparatif de l'état de l'art.

Chapitre 2

Contrôle de flux d'information par utilisation conjointe d'analyse statique et dynamique

2.1 Hypothèses et architecture générale

L'objet de ce chapitre est de présenter les hypothèses de départ, les problématiques, la méthodologie et l'approche utilisée pour réaliser du suivi de flux d'information multiniveau dans les différentes couches du système. Nous présentons d'abord les aspects liés au matériel puis ceux relatifs au logiciel.

2.1.1 Hypothèses et architecture matérielles

Le projet HardBlare cible en priorité les systèmes embarqués, mais l'approche proposée pourrait convenir également à d'autres types de systèmes (par exemple, des serveurs ou des ordinateurs de bureau). La volonté du projet est de permettre la démocratisation d'un système de contrôle de flux d'information précis, flexible et qui puisse prendre en compte les conteneurs d'informations des différentes couches du système. Pour cela, une des hypothèses de départ est que le matériel nécessaire pour réaliser ce projet doit être disponible dans le commerce. Il est donc nécessaire de s'adapter aux architectures et solutions déjà existantes. C'est pourquoi l'approche off-core a été retenue, puisqu'aucune modification du CPU n'est nécessaire et que le moniteur de sécurité est externalisé, laissant ainsi le choix aux fabricants de systèmes sur puce (System on Chip (SoC)) d'incorporer directement la solution dans le SoC ou de l'implémenter dans un circuit logique programmable. En effet, l'intégration de plus en plus courante d'une partie FPGA dans les

systèmes sur puce récents pour y intégrer des accélérateurs matériels pour la sécurité, l'intelligence artificielle, ou le traitement d'images [1] ouvre la voie à des solutions de sécurité qui ont un potentiel d'adoption et de démocratisation rapides.

Implémenter le moniteur de sécurité dans un composant externe, qu'il s'agisse d'un FPGA ou d'un coprocesseur intégré au SoC, permet de l'isoler du processeur principal (la cible) et de le protéger des attaques logicielles visant les applications exécutées sur le processeur principal. Toutefois, cela créer également un fossé sémantique [64] : du fait de l'isolation, le moniteur peut n'avoir qu'une vue partielle de l'état de la cible qu'il surveille. Dans notre cas, le moniteur doit avoir accès à des informations lui permettant de suivre l'intégralité des flux d'information. Cela nécessite peu ou prou d'avoir accès à l'ensemble des instructions exécutées par la cible. Il faut donc établir des canaux de communication entre la cible et le moniteur pour que ce dernier puisse accéder à ces informations. L'hypothèse principale du projet étant de ne pas modifier le processeur principal, la solution proposée doit reposer sur des moyens de communication existants. Le projet HardBlare s'intéresse notamment à l'utilisation des mécanismes matériels de débogage et de traces intégrés par les fabricants dans les CPU modernes, par exemple ARM CoreSight (ETM et Program Trace Macrocell (PTM)) [65], Intel PT [66] ou le mécanisme de trace des processeurs RISC-V [67].

Toutefois, les mécanismes de trace n'offrent généralement qu'une vue parcellaire de la trace d'exécution. Les solutions qui exportent l'ensemble des instructions exécutées par le processeur principal, par exemple ARM ETM [68], ne sont disponibles que pour des processeurs qui disposent d'une faible puissance de calcul, typiquement des microcontrôleurs destinés au marché de l'embarqué. En effet, cette approche génère un trafic de trace important et n'est pas envisageable pour les processeurs plus puissants. La plupart des solutions, telles ARM PTM [69] ou Intel PT, ne fournissent dans la trace que les informations relatives aux branchements et exceptions. L'ensemble des instructions exécutées entre deux sauts doit être inféré à partir du code machine du programme exécuté. En outre, les adresses de certains accès mémoires réalisés par le programme surveillé peuvent difficilement être inférées à partir du seul code du programme : en effet, certains peuvent dépendre du contexte d'exécution. Cela nécessite de mettre en place des canaux de communication supplémentaires entre le processeur principal et le moniteur.

Afin d'implémenter notre approche, nous avons choisi le système sur puce Xilinx Zynq-7000, qui comprend une partie système (Processing Sys-

1. https://www.intel.fr/content/www/fr/fr/products/programmable/fpga/stratix-10.html

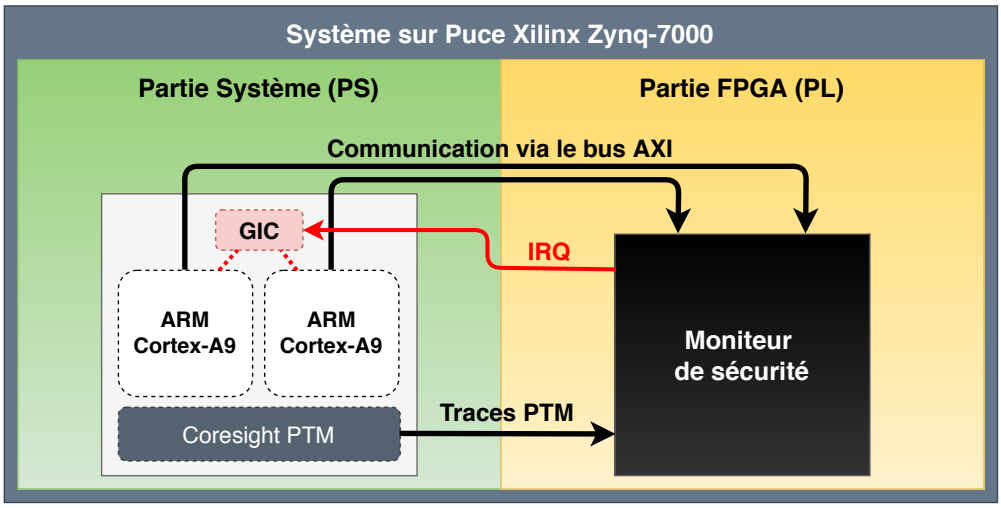


FIGURE 2.1 – Choix de l'architecture matérielle.

tem (PS)) incluant deux cœurs CPU ARM Cortex-A9, des composants de débogage ARM CoreSight PTM-A9 et une partie FPGA (Programmable Logic (PL)) dans laquelle le moniteur de suivi de flux d'information est implémenté. Le choix matériel est illustré par la figure 2.1. Les parties PS et PL sont connectées grâce au bus Advanced eXtensible Interface (AXI), qui permet de mettre en place des canaux de communications supplémentaires.

Nous souhaitons utiliser un moniteur optimisé pour le DIFT. Pour cela, nous nous appuyons sur les travaux et les développements réalisés par Muhammad Abdul Wahab dans le cadre de sa thèse de doctorat [70], au sein du projet HardBlare. Il s'agit principalement de développements matériels sur FPGA ayant conduit à l'implémentation de deux blocs principaux sur la partie PL du SoC Zynq :

— Un décodeur matériel des traces PTM générées par le processeur principal (PS) ;
— Un cœur permettant de réaliser du DIFT.

Le cœur DIFT est un processeur spécifique qui, à partir de la trace décodée, récupère les annotations de chaque bloc de base et les exécute. L'exécution de ces annotations permet de suivre les flux d'information générés par l'exécution de l'application surveillée, en propageant les étiquettes associées aux conteneurs d'information de l'application. Les conteneurs d'informations gérés par le cœur DIFT correspondent à ceux gérés directement par le processeur principal : les registres du processeur et la mémoire vive. Le cœur DIFT permet d'associer des étiquettes à ces deux types de conteneurs :

— le cœur dispose d'une banque de registres (Tag Register File) permet-

tant de stocker, pour chaque registre du processeur principal, l'étiquette correspondante ;
— le cœur permet de stocker, dans une zone mémoire spécifique et protégée (c'est-à-dire inaccessible à l'application), les étiquettes correspondant aux données stockées en mémoire vive.

Nous supposons enfin que l'ensemble du matériel est de confiance et, par exemple, qu'il ne contient pas de cheval de Troie matériel [71]. En outre, nous supposons que l'attaquant ne dispose pas d'un accès physique au matériel lorsque celui-ci est opérationnel. Par conséquent, nous ne considérons pas les attaques physiques telles que l'injection de fautes ou les attaques par canaux auxiliaires. Ce type de menaces nécessite des contre-mesures orthogonales aux mécanismes de sécurité développés dans le projet HardBlare, telles que le test fonctionnel du matériel [72], l'obfuscation de l'architecture matérielle [73], le masquage ou la détection d'erreur par redondance [74].

2.1.2 Hypothèses et architecture logicielles

Côté logiciel, nous voulons recourir à un système d'exploitation multitâche et multi-utilisateur existant (par exemple, Linux, Windows ou MacOS) permettant d'exécuter des applications sans nécessiter de redévelopper l'OS et les applications. En effet, les approches qui s'appuient sur un OS spécifique nécessitent un effort conséquent de développement des applications et des pilotes de périphériques, ce qui en limite l'adoption.

L'objectif du projet HardBlare est de protéger les différentes applications utilisateurs. Nous supposons que les couches logicielles de bas niveau (*firmware*, OS, hyperviseurs), qui s'exécutent avec de hauts niveaux de privilèges, sont de confiance. En effet, les mécanismes de traces peuvent être configurés par des logiciels s'exécutant avec ces niveaux de privilèges. Cela suppose de s'assurer de l'intégrité du code ce ces logiciels (par exemple, en utilisant des mécanismes de vérification d'intégrité au démarrage de l'ordinateur [75] ou lors des mises à jour). Il est en outre nécessaire d'utiliser des techniques complémentaires afin d'éviter les vulnérabilités dans ces couches logicielles ou de prévenir l'exploitation de ces vulnérabilités. Ces techniques sont orthogonales à l'approche développée dans le projet HardBlare.

Puisque le moniteur ne peut pas connaître l'ensemble des instructions exécutées par le processeur, des annotations décrivant les flux d'information engendrés dans la couche matérielle lors de l'exécution des instructions-machine sont nécessaires. Dans l'approche que nous proposons, nous générons ces annotations lors de la compilation des applications avec la bibliothèque Low-Level Virtual Machine (LLVM), pour chaque bloc de base du programme. Ces annotations sont sauvegardées dans le fichier exécutable de l'applica-

tion, dans une section dédiée. Elles sont ensuite envoyées au moniteur DIFT lors du chargement des applications. Nous avons dû pour cela modifier le chargeur d'applications de Linux.

Il est parfois indispensable d'envoyer au moniteur DIFT des informations dont seule l'application utilisateur a connaissance lors de son exécution, par exemple lors d'accès mémoire à des adresses calculées dynamiquement. Pour cela, nous avons choisi d'utiliser l'instrumentation de l'application afin d'envoyer ces informations nécessaires au moniteur. Dans notre approche, cette instrumentation est également mise en œuvre lors de la compilation des applications.

Comme expliqué en section 2.1.1, le moniteur DIFT ne peut associer des étiquettes qu'aux conteneurs d'information gérés directement par le processeur : les registres et la mémoire vive. Toutefois, il est également important de pouvoir associer des étiquettes aux conteneurs d'informations gérés par le système d'exploitation, notamment les fichiers. Cela permet de surveiller les flux entre différentes applications, lorsqu'une application écrit dans un fichier qui est par la suite lu par une autre application. La notion de fichier n'est pas une abstraction gérée par le microprocesseur et la notion d'adresse mémoire n'est pas adaptée pour associer des étiquettes aux fichiers. Il faut pour cela que le suivi de flux d'information soit également supporté par le système d'exploitation et le système de fichier. Les étiquettes doivent être stockées de façon persistante dans le système de fichier. Ainsi, lors d'un redémarrage, l'état précédent des étiquettes du système est préservé. Un mécanisme de communication doit être mis en place entre l'OS et le moniteur DIFT afin que l'OS puisse fournir l'étiquette correspondant à un fichier dans le cas d'une lecture ou modifier cette étiquette en cas d'écriture. Dans le cadre de mes travaux, j'ai modifié RfBlare [2], une extension de sécurité permettant de réaliser du DIFT au niveau du noyau Linux, afin de l'intégrer au projet HardBlare et gérer les étiquettes associées aux fichiers.

La Figure 2.2 illustre les différentes contraintes en termes de stockage et de propagation des étiquettes dans le système. Le système doit être en mesure de propager les étiquettes dans de la mémoire partagée, lors de l'utilisation de moyens de communication interprocessus et dans les différentes structures de données du système d'exploitation. Cela permet alors d'exprimer des politiques de sécurité telles que bloquer toute fuite d'information via le réseau en vérifiant que les données écrites sur un *socket* réseau ne sont pas censées être confidentielles.

Les politiques de sécurité doivent être flexibles, et modifiables selon les souhaits de l'administrateur système. La représentation des niveaux de sé-

2. http://www.blare-ids.org/rfblare/

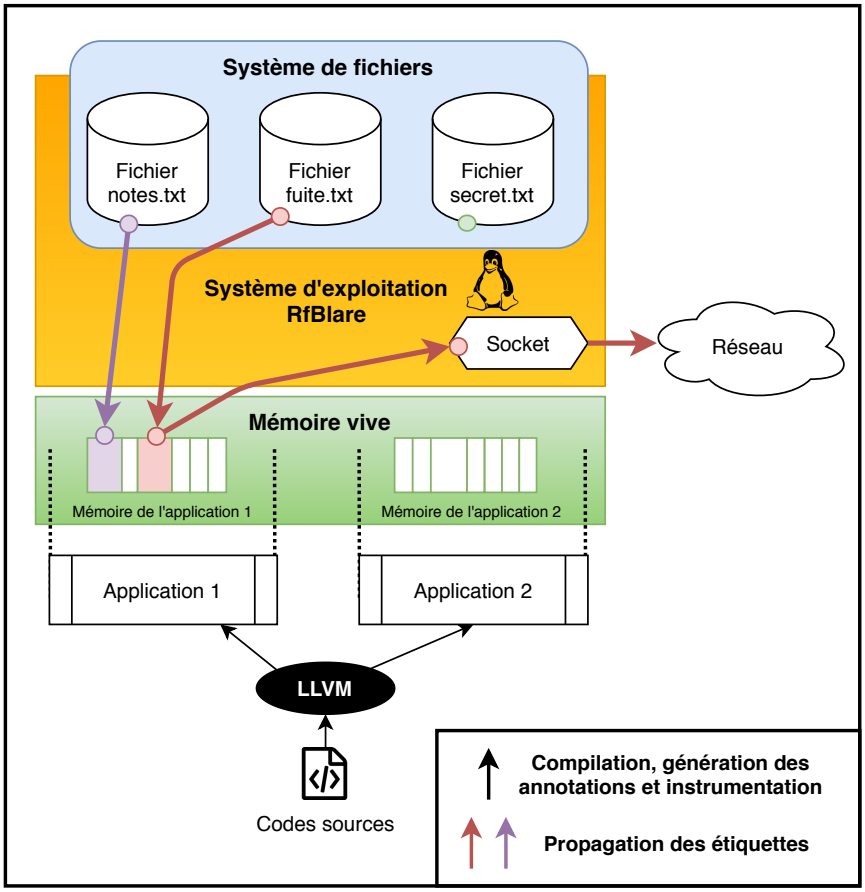


FIGURE 2.2 – Propagation des étiquettes dans les couches logicielles du système.

curité doit également être flexible avec une maîtrise de la granularité des données marquées. Un niveau de sécurité doit donc pouvoir être représenté avec une étiquette allant de 1 à 32 bits, permettant ainsi une représentation sous forme de treillis. Afin de détecter un grand nombre d'attaques grâce au suivi de flux d'information, plusieurs politiques de sécurité doivent pouvoir être vérifiées en même temps au niveau du moniteur. Lorsque le moniteur détecte une violation de politique de sécurité, le système d'exploitation doit en être informé le plus rapidement possible grâce à une interruption.

Cette volonté de propager les étiquettes dans tout le système, de la façon la moins intrusive possible, fait de HardBlare un projet ambitieux qui nécessite la modification de toutes les couches composant le système. De plus, ces modifications exigent une coordination entre les différents éléments composant le système.

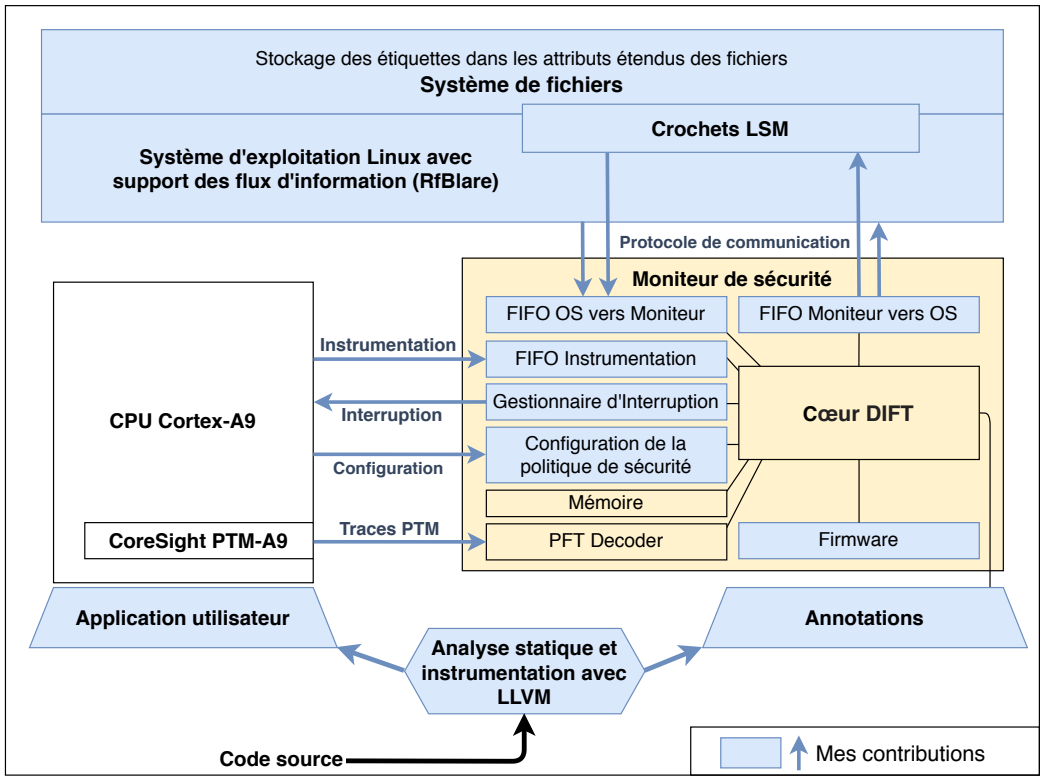


FIGURE 2.3 – Architecture globale et contributions.

La figure 2.3 présente l'architecture générale de la solution développée dans le cadre du projet HardBlare. Cette figure illustre également mes contributions sur ce projet :

- participation à la définition de l'architecture globale ainsi qu'au choix des technologies et des outils ;
- configuration et modification des pilotes Linux des composants ARM CoreSight PTM-A9 pour la génération de traces dans un contexte multitâche ;
- création d'une analyse statique dans LLVM permettant la génération des annotations et l'harmonisation du code avec les traces générées par les composants de débogage et de traces ;
- implémentation d'une phase d'instrumentation à l'aide de LLVM ;
- modification du noyau Linux et des crochets LSM pour la gestion du suivi de flux d'information ;
- développement du firmware du moniteur de sécurité et des protocoles de communication entre les différents composants ;
- développement du gestionnaire d'interruption du moniteur ;

— intégration du moniteur de sécurité et des différentes parties logiciels ;
— automatisation de la création d'une distribution Linux et d'une chaîne de compilation complète.

L'architecture repose avant tout sur l'utilisation de composants de débogage et de génération de traces. Sans ces composants, certaines informations dynamiques comme les interruptions ou les changements de flot de contrôle ne peuvent être connues. Il est donc nécessaire de configurer correctement ces composants afin de prendre en compte le contexte multithread et multitâche du système d'exploitation. Le composant doit s'activer lorsque l'application ciblée s'exécute sur le processeur et se désactiver lorsque l'ordonnanceur exécute une autre application sur le processeur.

2.2 Extraction passive de l'activité du processeur grâce aux composants de débogage et de traces du processeur

L'une des originalités du projet HardBlare est l'utilisation d'un composant de génération de traces afin d'extraire des informations concernant l'état du processus exécuté sur le CPU. Dans notre configuration matérielle, nous disposons d'un composant Coresight PTM-A9 développé par ARM, qui permet le suivi en temps réel de l'exécution de certaines instructions sans surcoût temporel.

Lorsque ce composant est activé et correctement configuré, il génère des traces à certains points appelés *waypoints*. Ces points sont relatifs aux modifications du registre PC et permettent donc de suivre le chemin pris par le processeur dans le graphe de flot de contrôle. L'hypothèse faite par les concepteurs du Coresight PTM-A9 est que le composant recevant les traces dispose également du code tracé, ce qui permet ainsi de corréler les traces au code de l'application. Des traces sont générées dès la rencontre des *waypoints* suivants :

— lors de l'exécution de branchements indirects et directs (la trace contient alors l'adresse cible ainsi que le résultat du code conditionnel) ;
— lorsqu'une exception ou une interruption est générée (la trace décrit alors les raisons de l'exception, l'adresse à laquelle cet évènement s'est produit, ainsi que l'adresse de retour dans le code) ;
— lorsque le jeu d'instruction ou l'état de sécurité du processeur change (la trace décrit alors les raisons et les conséquences de ce changement).

Afin d'illustrer le fonctionnement de ce mécanisme de traces, considérons l'exemple d'un programme ayant le code source 2.1. Ce programme lit le

n$^{\text{ième}}$ élément d'un tableau de 42 éléments. L'indice est fourni par l'utilisateur comme argument au lancement du programme, on ne peut donc pas se fier à l'indice donné puisqu'il peut être intentionnellement en dehors des bornes du tableau. C'est pourquoi la fonction `getValueFromArray` vérifie la valeur de l'indice pour s'assurer qu'elle ne soit pas supérieure à la taille du tableau. Le programmeur a malencontreusement oublié de vérifier que l'indice n'est pas négatif. En effet, si l'utilisateur donne un argument négatif, un accès mémoire en dehors de la borne inférieure du tableau est réalisé. Cette vulnérabilité est appelée *Out-of-bounds Read* et est désignée par la référence CWE-125 par The MITRE Corporation (MITRE).

```
0  #include <stdio.h>
1
2  int getValueFromArray(int *array, int len, int index)
3  {
4    int value;
5    if (index < len)
6      {
7        value = array[index];
8      }
9    else
10     {
11       value = -1;
12     }
13   return value;
14 }
15
16 int main(int argc, const char* argv[])
17 {
18   int array[42] = { 42 };
19   int x = atoi(argv[1]);
20   int result = getValueFromArray(&array, 42, x);
21   printf("Value is: %x\n", result);
22 }
```

Code source 2.1 – Exemple d'une application utilisateur vulnérable CWE-125

On peut constater que l'application affiche le résultat sur la sortie standard à l'aide de la fonction `printf`, qui appartient à la bibliothèque standard C. Si l'on souhaite identifier les appels système réalisés par l'application, ce qui est notre cas, ce type de fonction doit donc être également tracé puisqu'il contient un appel système pour permettre l'écriture sur la sortie standard.

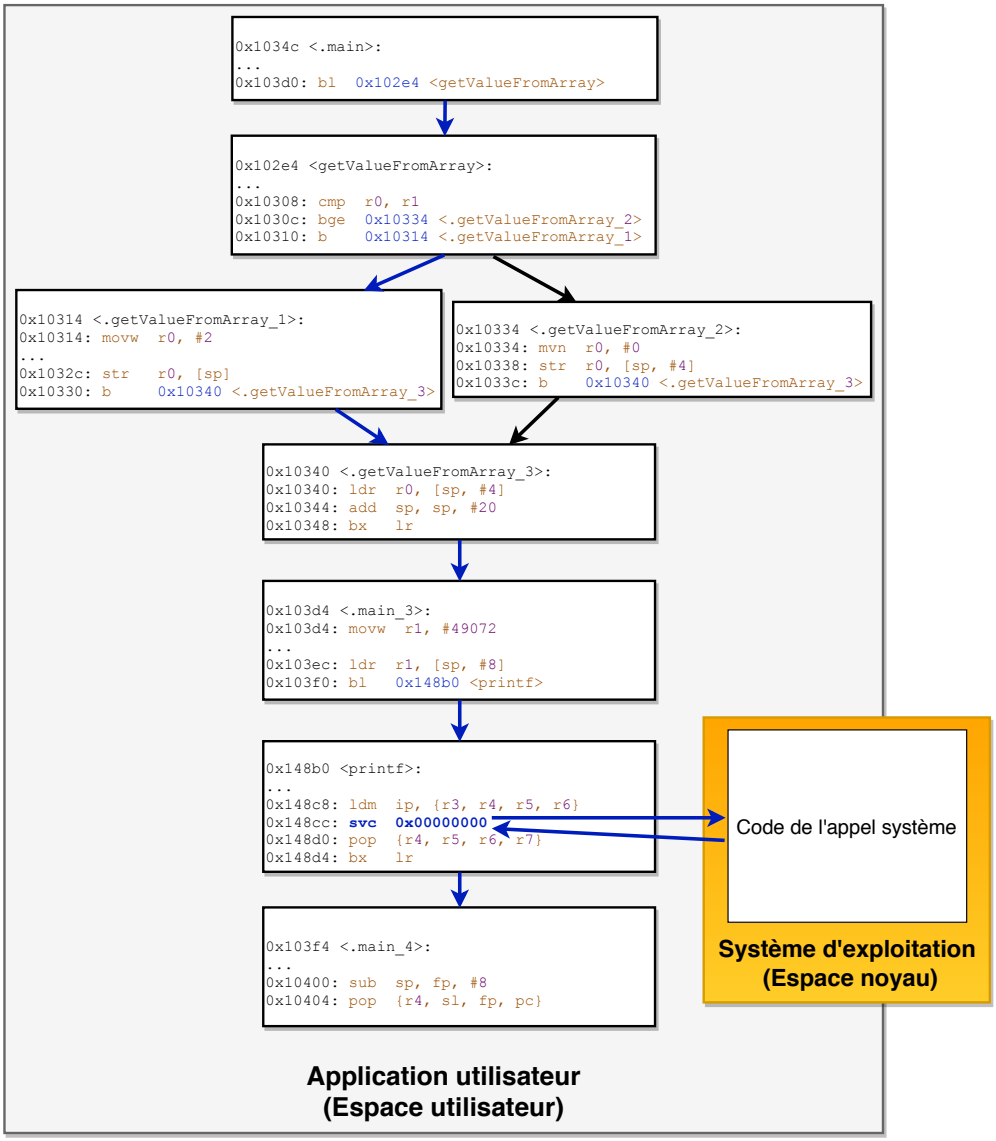


FIGURE 2.4 – Graphe de flot de contrôle.

La Figure 2.4 correspond au graphe de flot de contrôle obtenu après la compilation du programme. Des chemins différents peuvent être empruntés lors de l'exécution du programme et on ne peut savoir quel chemin sera pris avant l'exécution du programme, ce qui justifie le recours à une analyse dynamique. De plus, lors de l'exécution de l'instruction *svc* à l'adresse *0x148cc*, une exception est générée provoquant un branchement vers l'appel système concerné, ce qui fait passer le processeur du mode non-privilégié dans l'espace utilisateur, au mode privilégié dans l'espace noyau du système d'exploi-

tation. Une fois l'appel système terminé, l'application reprend à l'instruction suivante, c'est-à-dire à l'adresse *0x148d0*.

Lors de cet appel système, le processeur exécute du code présent dans le noyau du système d'exploitation. Le PTM ne doit donc plus générer de traces, car le système d'exploitation est considéré de confiance. Pourtant, des flux d'information et des modifications de l'état de l'application peuvent être réalisés lors d'un appel système. Puisque le code noyau n'est ni tracé ni annoté, il est nécessaire d'informer le moniteur de sécurité des effets de bords et des flux d'information produits lors d'un appel système. Par exemple, certains appels système ont un impact sur la mémoire de l'application ou viennent écraser la valeur de certains registres pour y inclure leurs données de retour. Dans le cas de notre exemple, un flux d'information est réalisé entre les données de la variable `result` et le fichier représentant la sortie standard. Durant l'exécution du programme, les traces générées par le PTM sont données dans le Code 2.2, le chemin correspondant est représenté en bleue dans la Figure 2.4. Nous utilisons donc la trace fournie par le PTM à la ligne 7 du Code 2.2 indiquant qu'un appel système s'est produit pour synchroniser le moniteur de sécurité et le système d'exploitation dans le but d'échanger des informations pour propager des étiquettes. En effet, le moniteur de sécurité doit envoyer l'étiquette de la variable `result` au système d'exploitation afin de propager l'étiquette vers le fichier représentant la sortie standard.

```
1 0x1034c (ARM state)
2 0x102e4 (ARM state)
3 0x10314 (ARM state)
4 0x10340 (ARM state)
5 0x103d4 (ARM state)
6 0x148b0 (ARM state)
7 0x148cc (Exception Supervisor Call, ARM state, Secure
     state)
8 0x148d0 (ARM state)
9 0x103f4 (ARM state)
```

CODE SOURCE 2.2 – Traces générées par le PTM-A9 lors de l'exécution de l'application utilisateur

Cette application est exécutée sur un système d'exploitation multitâche, elle est donc en concurrence avec d'autres applications. Il est donc nécessaire de filtrer les traces uniquement lorsque l'application ciblée est exécutée et d'arrêter le mécanisme de trace dès que l'ordonnanceur de l'OS change d'application. On peut ainsi tirer profit de ces traces et les envoyer au moniteur de sécurité pour que celui-ci puisse les traiter. Pour cela il est nécessaire de

configurer les composants de débogage et de traces. Ma contribution sur ce point a été de filtrer les traces à l'aide du Process IDentifier (PID) de l'application, pour générer uniquement des traces lorsque l'application tracée est exécutée sur le processeur. Pour cela, une modification du pilote Linux des composants CoreSight a été nécessaire ainsi que la modification de la fonction `_start` dans la bibliothèque standard C car la valeur du PID n'est connue qu'à la création du processus. C'est pourquoi il est nécessaire de configurer le filtrage du PID et d'activer les composants CoreSight directement dans la bibliothèque standard C avant le démarrage de la fonction principale de l'application utilisateur.

Ces traces sont compressées et envoyées par paquets vers le Trace Port Interface unit (TPIU) pour pouvoir être envoyées vers le moniteur de sécurité situé dans la PL à l'aide du bus Extended Multiplexed Input Output (EMIO). La forme des traces suit la spécification du Program Flow Trace (PFT) [69]. Ces traces sont ensuite décompressées et décodées à l'aide du PFT Decoder, dont le développement a été réalisé par Muhammad Abdul Wahab [76], puis envoyées dans une mémoire First In, First Out (FIFO). La Figure 2.5 illustre la configuration des composants de débogage et de traces dans un environnement multitâche et le chemin emprunté par les traces jusqu'au moniteur de sécurité.

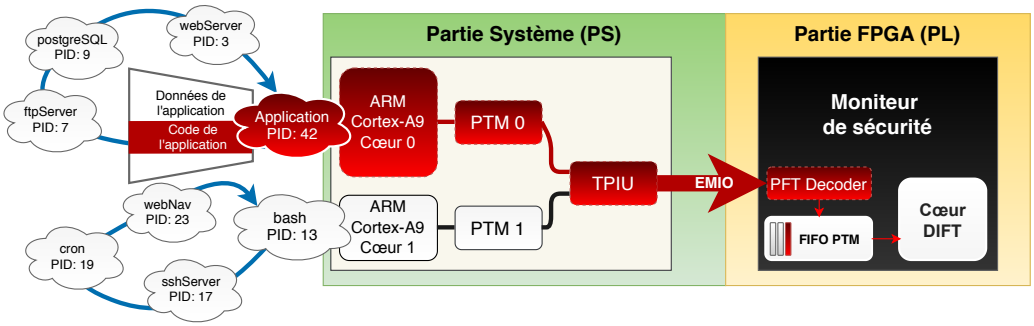


FIGURE 2.5 – Configuration des composants de débogage et de traces.

Bien que ces traces soient très utiles, elles restent néanmoins insuffisantes pour pouvoir être utilisées pour le suivi de flux d'information. En effet, le PTM fournit des traces uniquement lors de l'exécution de *waypoints* afin de minimiser la bande passante sur le bus EMIO. Aucune information sur les flux d'informations entre deux traces n'est fournie. On ne peut donc pas connaître les instructions qui y sont exécutées et donc la dissémination de l'information dans l'architecture. Il est ainsi nécessaire d'expliciter via des annotations les flux d'informations qui se produisent entre deux traces. Ces annotations sont générées grâce à une analyse statique lors de la compilation

de l'application, et feront l'objet de la Section 2.4.

De plus, on peut remarquer que toutes les adresses reçues par le PTM ne concordent pas forcément avec les adresses d'entrée des blocs de base du programme. Par exemple, l'instruction *svc* à l'adresse *0x148cc* de la Figure 2.4 est au plein milieu d'un bloc de base. Il est donc nécessaire d'identifier les différences entre la définition d'un bloc de base par le compilateur et la définition d'un *waypoint* défini par le composant PTM. Les annotations doivent être organisées selon les traces générées à l'aide du PTM. Par conséquent, une transformation du code de l'application lors de la compilation est indispensable afin d'obtenir une correspondance entre les bloc de base et les trace. Cette transformation fera l'objet de la Section 2.3.

2.3 Organisation des annotations et transformation de l'application en vue de son harmonisation avec les traces générées

Bien que le processeur Cortex-A9 dispose de plusieurs jeux d'instructions tel que *Thumb*, notre implémentation ne les prend pas en charge. En effet, la prise en charge de différents formats d'instructions rend plus complexe le décodage des traces du composant PTM et la génération des annotations. Certaines instructions permettent de basculer d'un jeu d'instruction à un autre. Par exemple l'instruction *bx* permet d'effectuer un saut vers une adresse cible et de permuter le jeu d'instruction de *ARM* vers *Thumb* ou de *Thumb* vers *ARM*. Dans notre approche, nous supposons que ces instructions ne doivent pas être présentes dans le code de l'application. C'est pourquoi, la première adaptation requise est de forcer le compilateur à produire uniquement des instructions issues de l'Instruction Set Architecture (ISA) *ARM*. Pour cela, une modification au niveau du *back-end* du compilateur a été nécessaire.

Le jeu d'instruction *ARM* offre la possibilité d'exécuter la plupart des instructions de façon conditionnelle. Par exemple dans le Code 2.3, l'instruction à l'adresse *0x100* compare les valeurs des registres $r0$ et $r1$ et stocke le résultat de cette comparaison dans les champs du registre Application Program Status Register (APSR). Si le champ *APSR.Z* est égal à un, ce qui signifie que les valeurs des registres comparés *r0* et *r1* sont égales, alors l'instruction à l'adresse *0x104* est exécutée sinon elle équivaut à un No Operation (nop). L'instruction à l'adresse *0x108* est quant à elle exécutée uniquement si le champ *APSR.Z* est à zéro, c'est-à-dire si les valeurs des registres *r0* et *r1* ne sont pas égales. Pour finir, l'instruction de branchement à l'adresse

0x10C est exécutée dans les mêmes conditions que les instructions précédentes. Elle dépend elle aussi du résultat de l'instruction de comparaison à l'adresse *0x100*.

```
0x100  cmp r0, r1
0x104  moveq r0, r2
0x108  movne r0, r3
0x10C  beq 0x1234
```

CODE SOURCE 2.3 – Exemple d'instructions ARM utilisant les codes conditionnels

Le PTM produit des traces exclusivement lors de l'exécution de *waypoints*. Puisque les instructions conditionnelles situées aux adresses *0x100* à *0x108* dans l'exemple précédent ne sont pas des *waypoints*, aucune trace n'est générée par le PTM pour ces instructions. Lors de l'exécution du programme, l'analyse des traces produites par le PTM ne permet donc pas de déterminer si ces instructions ont été exécutées et donc s'il est nécessaire de propager les étiquettes entre les conteneurs d'information utilisés dans ces instructions. Par contre, l'instruction de branchement conditionnel située à l'adresse *0x10C* génère une trace PTM. Dans notre approche, nous supposons donc que le code de l'application ne contient pas d'instruction conditionnelle, exceptée les formes conditionnelles des instructions de branchement.

La deuxième adaptation a été la modification du *back-end* du compilateur pour désactiver le recours à des formes conditionnelles pour les instructions qui ne sont pas des *waypoints*. Cette restriction n'est en pratique pas très pénalisante [77]. C'est également la décision qui a été prise par ARM pour les versions suivantes de leur architecture, à savoir l'ARMv8 (64 bit) [77] : « *The A64 instruction set does not include the concept of predicated or conditional execution. Benchmarking shows that modern branch predictors work well enough that predicated execution of instructions does not offer sufficient benefit to justify its significant use of opcode space, and its implementation cost in advanced implementations.* ».

La définition d'un bloc de base par LLVM est un bloc contenant une suite d'instructions qui s'exécutent de façon séquentielle et se termine par un *terminator*. Un *terminator* est une instruction de branchement intraprocédurale. Les instructions utilisées pour réaliser un appel système (instruction *svc*) ou un appel à une fonction (branchement interprocédurale) ne sont donc pas considérées comme des *terminator* puisque l'exécution reprend au même endroit dans le bloc de base, une fois l'appel terminé. Toutefois, les adresses

émises par les traces PTM ne correspondent pas exactement à cette définition. En effet, le PTM génère une trace pour chaque *waypoint* exécuté. Il est donc nécessaire de découper l'application selon la notion de *waypoint* définie par le composant de débogage et de traces.

Lorsqu'une instruction utilisée pour réaliser un appel système (*svc*) ou un appel à une fonction externe (branchement interprocédurale) est rencontrée et que celle-ci n'est pas la dernière instruction du bloc de base, il est essentiel de différencier les annotations relatives aux instructions qui la succèdent pour ne pas propager leurs flux d'information. Pour cela, un symbole est créé et ajouté juste après ladite instruction. La valeur de ce symbole correspondra donc à la trace générée par le PTM lors du retour de l'appel système ou de la fonction appelée et permettra alors de faire référence à la liste d'annotations à exécuter par le moniteur de sécurité une fois l'appel retourné.

L'algorithme 1 présente l'analyse statique responsable de l'organisation des annotations selon les traces générées lors de l'exécution des *waypoints*. Chaque adresse correspondant à une trace générée par le PTM possède une liste d'annotations décrivant les flux d'information qui se produisent jusqu'à la prochaine trace.

Algorithme 1 : Organisation des annotations par *waypoints*.

```
pour chaque Fonction F faire
    pour chaque Bloc de base BB faire
        Sym = symbole du bloc de base BB
        ListeAnnotations{Sym} = ()
        pour chaque Instruction I faire
            Ajouter l'annotation relative à I dans
             ListeAnnotations{Sym}
            si I est un waypoint alors
                si I n'est pas la dernière instruction dans le bloc de
                 base BB alors
                    NewSym = Insérer un symbole après l'instruction
                     I
                    Sym = NewSym
                    ListeAnnotations{Sym} = ()
        fin
    fin
fin
```

La Figure 2.6 illustre ce cas, le PTM va générer une trace contenant l'adresse *0x10300* lors de l'entrée dans le bloc de base, suivi d'une trace contenant l'adresse de l'instruction affectant le flot de contrôle (*0x10304*), puis

lors du retour de l'appel, une dernière trace est générée contenant l'adresse *0x10308*. Grâce à cette troisième adaptation, les traces pourront être correc-

(a) Avant transformation.

(b) Après transformation.

FIGURE 2.6 – Instruction affectant le flot de contrôle au milieu d'un bloc de base.

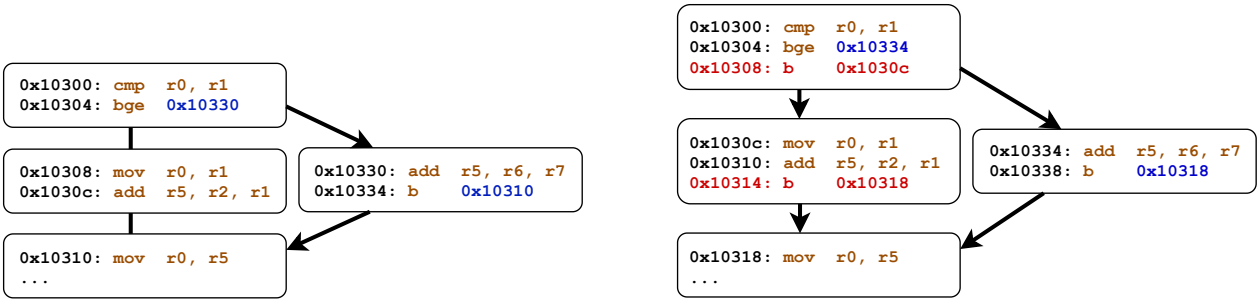
tement interprétées sans sur approximation. Ainsi, les annotations de chaque adresses reçu par le PTM sont alors exécutées dans le même ordre d'exécution que les instructions ARM dans le processeur.
La quatrième adaptation à réaliser est l'ajout d'une instruction de branchement inconditionnel à la fin de chaque bloc de base qui continue implicitement vers un bloc de base sans instruction de branchement. En effet lorsque deux blocs de base se suivent dans le code de l'application, l'incrément naturel du registre PC permet de passer implicitement d'un bloc de base à un autre. Cette transformation présentée par l'algorithme 2 permet alors de forcer le PTM à générer une trace à la sortie de chaque bloc de base que le processeur exécute.

Dans l'exemple de la figure 2.7, avant la transformation, le PTM génère une trace contenant l'adresse du premier bloc de base *0x10300*. Si le branchement à l'adresse *0x10304* est pris, alors les traces contenant l'adresse *0x10330* puis *0x10310* sont générées. Le coprocesseur DIFT peut donc exécuter les annotations correspondant à ces blocs de base. En revanche, si le branchement n'est pas pris à l'adresse *0x10304*, le coprocesseur ne reçoit pas les adresses des blocs de base situés aux adresses *0x10308* et *0x10310*, qui sont exécutés par la suite. Il ne peut donc pas exécuter les annotations correspondant à ces blocs de base. Dans le code transformé, les sauts directs ajoutés aux adresses *0x10308* et *0x10314* génèrent des traces permettant au coprocesseur d'identifier les blocs de base exécutés à la suite du premier, si le branchement n'est pas pris.

Une première étape de ma contribution a donc été l'élaboration et l'implémentation de ces adaptations et transformations lors de la compilation du programme utilisateur. Les annotations propageant les étiquettes des conteneurs d'information dans l'architecture matérielle peuvent ainsi être regroupées et référencées par les adresses données dans les traces PTM.

Algorithme 2 : Analyse statique permettant l'ajout d'un branchement inconditionnel à la fin de chaque bloc de base continuant implicitement vers le bloc de base suivant.

pour chaque *Fonction F* **faire**
  pour chaque *Bloc de base BB* **faire**
    pour chaque *Instruction I* **faire**
      si *I est la dernière instruction dans le bloc de base BB* **alors**
        si *le bloc de base BB continue implicitement sur un bloc de base BBfallThrough* **alors**
          Ajouter une instruction de saut inconditionnel vers le bloc de base *BBfallThrough* après l'instruction *I*
    fin
  fin
fin



(a) Avant transformation.

(b) Après transformation.

FIGURE 2.7 – Bloc de base se succédant sans branchement.

La seconde étape de ma contribution consiste à élucider le problème suivant : **Quels sont les flux d'information qui se produisent pour chaque trace PTM ?** Pour répondre à cette question, une analyse statique du code machine de l'application doit être réalisée afin de déduire les flux d'information qui se produisent entre les différents conteneurs d'information de l'architecture matérielle, à savoir les registres CPU et les adresses mémoires. Pour pouvoir décrire les flux d'information qui se produisent dans l'architecture matérielle, nous devons d'abord identifier les différents conteneurs d'information présents dans le processeur ARM Cortex-A9.

2.4 Flux d'information dans l'architecture matérielle du système

L'objet de cette section est de poser les bases permettant la création d'annotations décrivant les flux d'information à grain fin engendrés dans l'architecture matérielle. Pour cela il est nécessaire d'identifier les conteneurs d'informations présents dans le processeur et les différents types de flux d'information qui s'y produisent. La Section 2.4.1 introduit succinctement l'architecture du processeur ARM Cortex-A9, identifie les différents conteneurs d'information présents dans le processeur et les étiquettes correspondantes. Nous expliquons ensuite comment les annotations sont générées pour les instructions générales dans la Section 2.4.2. Nous exposons l'analyse statique utilisée pour gérer les flux implicites dans la Section 2.4.4. Pour finir, nous présentons la génération d'annotations pour les instructions manipulant la mémoire dans la Section 2.4.3.

2.4.1 Représentation des étiquettes pour les différents conteneurs d'information inhérents à l'architecture matérielle.

Le processeur Cortex-A9 est un processeur multicœur 32 bits développé par ARM et reposant sur l'architecture ARMv7-A, de type Reduced Instruction Set Computer (RISC). Lors de l'exécution d'un programme sur le processeur, douze registres généraux de 32 bits notés de $r0$ à $r8$ et de $r10$ à $r12$ peuvent être utilisés par le code des applications pour stocker des données. Nous avons réservé le registre $r9$ pour l'utilisation de l'instrumentation, dont le principe est détaillé dans la Section 2.5. Ces registres processeur ont donc chacun une étiquette représentant leur niveau de sécurité, qui est stockée dans le moniteur de sécurité que l'on notera de $\underline{r0}$ à $\underline{r8}$ et de $\underline{r10}$ à $\underline{r12}$. Le processeur Cortex-A9 est pourvu de trois registres spéciaux qui sont réservés à une utilisation prédéfinie : les registres Stack Pointer (SP), Link Register (LR) et PC.

Le registre SP noté $r13$ ou sp permet de stocker l'adresse du somment de la pile. Sa valeur correspond toujours à une adresse mémoire. Ce registre est souvent utilisé dans le code de l'application pour référencer les variables de l'application présentes sur la pile. Ce registre possède une étiquette notée $\underline{sp}$.

Le registre LR, noté $r14$ ou lr, est utilisé pour stocker l'adresse de retour depuis une sous-routine. Ce registre ne doit donc pas être utilisé par le développeur d'une application, seul le compilateur l'utilise pour pouvoir effectuer un retour vers la fonction appelante. Ce registre possède une étiquette notée

$\underline{lr}$.

Le registre PC noté $r15$ ou pc correspond au registre contenant l'adresse de la prochaine instruction à exécuter sur le processeur. La valeur de ce registre n'est pas intéressante pour le moniteur de sécurité car le mécanisme de traces PTM nous fournit les valeurs de ce registre à chaque exécution de *waypoints*. Néanmoins, durant l'exécution d'une application, des branchements conditionnels permettent de choisir l'adresse cible des instructions à exécuter sur le processeur. L'adresse choisie sera stockée dans le registre PC. Ces choix sont guidés par la vérification de conditions présentes dans le registre APSR et qui seront détaillées plus tard. Un flux d'information implicite existe donc entre la condition du branchement et les instructions qui seront par la suite exécutées. Puisque chaque condition influence l'exécution d'un nombre limité d'instructions, il est alors primordial de pouvoir restaurer l'étiquette précédente une fois que la condition n'a plus d'impact sur l'exécution des instructions. Une pile est donc la meilleure structure de donnée afin de représenter ces étiquettes.

La pile d'étiquettes du registre PC est notée $\underline{\widehat{PC}}$. Dans les définitions qui suivent, on suppose que $\underline{\widehat{PC}}$ contient n étiquettes avant l'application de chaque fonction :

$$\underline{\widehat{PC}} = (e_1, \ldots, e_n)$$

La fonction $push_{\underline{\widehat{PC}}}(e)$ empile l'étiquette donnée en paramètre sur la pile $\underline{\widehat{PC}}$.

$$push_{\underline{\widehat{PC}}}(e) \rightarrow \underline{\widehat{PC}} = (e_1, \ldots, e_n, e) \tag{2.1}$$

La fonction $pop_{\underline{\widehat{PC}}}()$ dépile l'étiquette du bloc de base actuellement traité et la retourne.

$$pop_{\underline{\widehat{PC}}}() = e_n \rightarrow \underline{\widehat{PC}} = (e_1, \ldots, e_{n-1}) \tag{2.2}$$

La fonction $top_{\underline{\widehat{PC}}}()$ renvoie l'étiquette du bloc de base actuellement traité sans désempiler.

$$top_{\underline{\widehat{PC}}}() = e_n \rightarrow \underline{\widehat{PC}} = (e_1, \ldots, e_n) \tag{2.3}$$

La Figure 2.8 illustre la propagation des étiquettes des conditions vers l'étiquette du registre PC et la représentation des étiquettes du registre PC sous forme de pile. On peut remarquer que la condition de branchement `foo < 0` à la fin du bloc de base *BB1*, influe sur le choix des prochaines instructions à exécuter, à savoir le bloc de base *BB2* ou *BB5*. L'étiquette de la variable `foo` (qui a un niveau de sécurité bleu) doit donc être combinée avec l'étiquette du haut de la pile et empilée après le branchement. De même, lorsque le bloc de base *BB2* est traversé, la condition `bar > 0` permet d'exécuter les

instructions du bloc de base *BB3* ou *BB4*. Ces deux blocs de base dépendent donc à la fois de la condition sur la variable `foo` et `bar`. L'étiquette du registre PC dans les blocs de base *BB3* et *BB4* correspond au mélange de l'étiquette par défaut du registre PC (de couleur blanche), de la variable `foo` (de couleur bleue) et de la variable `bar` (de couleur rouge). Ce mélange donne du violet.

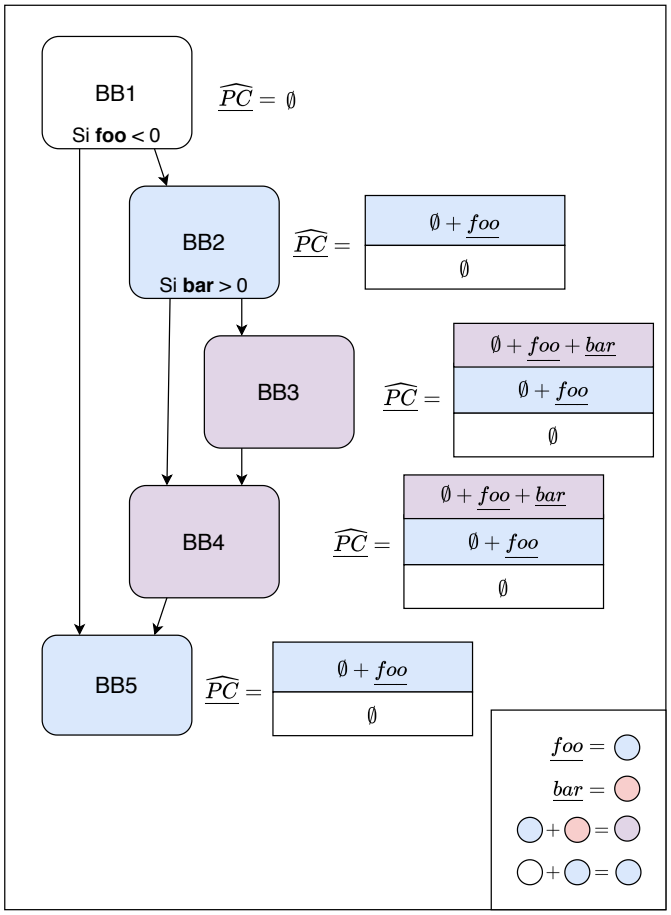


FIGURE 2.8 – Étiquettes du registre PC lors de l'exécution du programme.

L'interrogation qui se pose alors est : **quand et à quel endroit faut-il empiler et désempiler les étiquettes du registre PC ?** Pour répondre à cette question, une analyse statique a été développée et sera détaillée dans la Section 2.4.4.

Le registre de statut APSR est un registre constitué de champs utilisés par les instructions de comparaisons et les codes conditionnels. Ce registre influence l'exécution de certaines instructions et contient les champs suivants :

Le champ N pour « Négatif », prends la valeur 1 si le résultat de l'instruction est négatif.

Le champ Z pour « Zéro », prend la valeur 1 si le résultat de l'instruction est égal à 0.

Le champ C permet de constater que l'opération a produit une retenue (par exemple lors d'une addition).

Le champ V permet de détecter lorsque l'instruction a produit un dépassement de capacité.

Le champ GE est utilisé par les instructions relatives aux médias afin de détecter si les valeurs sont plus grandes ou égales.

Le champ Q permet de détecter une saturation, pour les opérations arithmétiques saturées

Chaque champ du registre APSR possède une étiquette, notée respectivement $\underline{APSR.N}$, $\underline{APSR.Z}$, $\underline{APSR.C}$, $\underline{APSR.V}$, $\underline{APSR.GE}$ et $\underline{APSR.Q}$.

Les instructions possédant une forme conditionnelle sont exécutées seulement si la condition est vérifiée. Lorsque la condition n'est pas vérifiée, l'instruction est interprétée comme une instruction `nop` par le processeur. Cette condition est ajoutée sous forme de suffixe à l'instruction. Le Tableau 2.1 présente les différents suffixes utilisés pour les codes conditionnels, ainsi que les étiquettes impliquées.

En annotant les flux d'information des codes conditionnels, on permet la propagation des étiquettes afin de prendre en compte les flux d'information implicites qui se produisent dans un bloc de base possédant un branchement conditionnel.

Tous les registres CPU précédemment présentés possèdent une étiquette stockée par le moniteur de sécurité dans une mémoire dédiée appelée Tag Register File (TRF).

En plus de ces registres, l'application a accès à la mémoire vive du système pour bénéficier davantage d'espace mémoire. En conséquence, les mots mémoires doivent également être traités comme des conteneurs d'information. En effet, l'application génère des flux d'information lors de la lecture ou de l'écriture de données depuis la mémoire vive.

La plupart des processeurs, et notamment le processeur ARM Cortex-A9, disposent d'une unité de gestion de la mémoire appelée Memory Management Unit (MMU) qui est chargée de traduire les adresses virtuelles en adresses physiques et de protéger l'accès à ces pages à l'aide d'un contrôle d'accès. Dans cette thèse, nous considérons que l'application tracée fonctionne avec un système d'exploitation qui utilise la pagination et la MMU. L'application tracée accède donc à la mémoire à l'aide d'adresses virtuelles. Notre approche associe donc exclusivement des étiquettes aux adresses virtuelles.

De plus, le jeu d'instruction des processeurs ARM est basé sur le principe du *Load-Store Architecture*. On peut dans ce cas distinguer les instructions

Suffixe	**Signification**	**Champs du registre APSR vérifiés**	**Étiquettes impliquées**
eq	Égal	$Z == 1$	$\underline{APSR.Z}$
ne	Non égal	$Z == 0$	$\underline{APSR.Z}$
cs ou **hs**	Retenue générée	$C == 1$	$\underline{APSR.C}$
cc ou **lo**	Pas de retenue générée	$C == 0$	$\underline{APSR.C}$
mi	Négatif	$N == 1$	$\underline{APSR.N}$
pl	Positif ou égale à zero	$N == 0$	$\underline{APSR.N}$
vs	Dépassement de capacité sur une valeur signée	$V == 1$	$\underline{APSR.V}$
vc	Pas de dépassement de capacité sur une valeur signée	$V == 0$	$\underline{APSR.V}$
hi	Version non signé de « Plus grand que »	$(C == 1)$ && $(Z == 0)$	$\underline{APSR.C}, \underline{APSR.Z}$
ls	Version non signé de « Plus petit que »	$(C == 0)$ || $(Z == 1)$	$\underline{APSR.C}, \underline{APSR.Z}$
ge	Version signé de « Plus grand ou égal »	$N == V$	$\underline{APSR.N}, \underline{APSR.V}$
lt	Version signé de « Plus petit que »	$N! = V$	$\underline{APSR.N}, \underline{APSR.V}$
gt	Version signé de « Plus grand que »	$(Z == 0)$ && $(N == V)$	$\underline{APSR.Z}, \underline{APSR.N}, \underline{APSR.V}$
le	Version signé de « Inférieur ou égal que »	$(Z == 1)$ || $(N! = V)$	$\underline{APSR.Z}, \underline{APSR.N}, \underline{APSR.V}$

TABLE 2.1 – Codes conditionnels.

qui accèdent à la mémoire, que l'on notera *MEMORY_INSTR*, et les instructions qui n'ont eux aucun effet de bord sur la mémoire, que l'on notera *GENERAL_INSTR*. Les instructions de type *MEMORY_INSTR* permettent seulement de charger des données depuis la mémoire vers des registres ou d'écrire des données présentes dans des registres vers la mémoire. Les transferts directs entre deux données présentes en mémoire sont donc impossibles.

Pour pouvoir accéder à la mémoire, le jeu d'instruction ARM oblige le stockage de l'adresse cible dans un des registres du processeur. Cette manière d'accéder à la mémoire à l'aide d'un registre processeur se nomme l'adressage indirect. Pour pouvoir étiqueter la mémoire, le moniteur de sécurité doit donc être capable de connaître la valeur du registre qui contient l'adresse cible. Supposons que le registre *r1* contienne la valeur hexadécimale *0xb6fb0024* représentant une adresse. Deux cas peuvent se présenter, soit le moniteur de sécurité a connaissance de cette valeur, soit l'application doit envoyer cette valeur au moniteur de sécurité grâce à l'instrumentation de l'application détaillée dans la Section 2.5.

La fonction *value(registre)* renvoie la valeur contenue dans le registre CPU passé en paramètre. Deux cas peuvent se présenter, soit le moniteur de sécurité a connaissance de cette valeur et la retourne, soit le moniteur doit récupérer cette valeur à l'aide de la fonction *instrumentation(registre)*.

$$value(r1) = \begin{cases} 0xb6fb0024 & \text{si } la\ valeur\ est\ connue \\ instrumentation(r1) = 0xb6fb0024 & \text{sinon.} \end{cases} \tag{2.4}$$

La fonction *address* permet d'interpréter une valeur comme une adresse. Le résultat renvoyé est donc un pointeur référençant une adresse mémoire.

$$\begin{aligned} address(value(r1)) &= address(0xb6fb0024) \\ &= *(0xb6fb0024) \end{aligned} \tag{2.5}$$

L'étiquette d'une adresse a la même notation que pour un registre.

$$\begin{aligned} \underline{(address(value(r1)))} &= \underline{(address(0xb6fb0024))} \\ &= \underline{*(0xb6fb0024)} \end{aligned} \tag{2.6}$$

Ainsi, toutes les adresses de la mémoire vive utilisées par l'application sont étiquetées par le moniteur de sécurité dans une mémoire dédiée appelée Tag Memory (TM). À présent que les différents conteneurs d'information de l'architecture matérielle sont identifiés et qu'il est possible de les étiqueter, il est nécessaire de comprendre comment ils sont utilisés par les instructions de l'ISA ARM et quels flux d'information en découlent. La Section 2.4.2 présente les différents types d'instructions de type $GENERAL\_INSTR$ et leur traitement permettant la génération d'annotations. La Section 2.4.3 traite des instructions de type $MEMORY\_INSTR$ permettant la lecture et l'écriture de données depuis la mémoire de l'application et les annotations correspondantes.

2.4.2 Annotations pour les instructions générales

Les instructions prennent comme opérandes soit des registres CPU présentés précédemment, soit des valeurs immédiates qui sont présentes directement dans le code de l'application. Il est donc nécessaire d'affecter une étiquette par défaut pour toutes les valeurs immédiates présentes dans le code, que l'on notera $\underline{Imm}$. La valeur de l'étiquette pourra ensuite être fixée librement par l'administrateur système.

Supposons une instruction qui manipule deux opérandes sources a et b et écrit le résultat de l'opération vers un registre destination c. Puisque a, b et c

sont des conteneurs d'information, une étiquette leur est alors associée, notée $\underline{a}$, $\underline{b}$ et $\underline{c}$. Dans le but de proposer des politiques de sécurité flexibles, la façon dont les étiquettes correspondant aux opérandes sont mixées entre elles est définie par l'opérateur $\underset{T}{\oplus}$, où T correspond au type de l'instruction produisant le flux d'information. Les différents types d'instructions seront présentés tout au long de cette section. Lorsqu'une instruction produit un flux d'information de ces opérandes vers le registre destination, il est nécessaire de propager les étiquettes des opérandes vers celle du registre destination, comme décrite par la Formule 2.7 :

$$\underline{a} \underset{T}{\oplus} \underline{b} \rightarrow \underline{c} \tag{2.7}$$

Nous détaillons par la suite les annotations générées pour chaque type d'instruction.

Les instructions arithmétiques et logiques permettent d'effectuer des opérations uniquement sur les douze registres généraux (de $r0$ à $r8$ et de $r10$ à $r12$). Ce groupe d'instructions, nommé $ARITH\_LOGIC$, à la forme présentée par la Formule 2.8, avec OP le nom de l'instruction, Rd le registre destination, Rn un registre source, et $Operand$ soit un deuxième registre source soit une valeur immédiate.

$$< OP >< Rd >< Rn >< Operand > \tag{2.8}$$

Ce type d'instruction génère un flux d'information entre les opérandes sources et l'opérande destination, l'annotation générée est donc :

$$\underline{Rn} \underset{ARITH\_LOGIC}{\oplus} \underline{Operand} \rightarrow \underline{Rd}$$

La Figure 2.9 présente un exemple d'instructions arithmétiques et logiques et les annotations générées pour la propagation des étiquettes.

```
add r0, r3, r4
or r5, r0, #0xFF00
```

$$\underline{r3} \underset{ARITH\_LOGIC}{\oplus} \underline{r4} \rightarrow \underline{r0}$$
$$\underline{r0} \underset{ARITH\_LOGIC}{\oplus} \underline{Imm} \rightarrow \underline{r5}$$

FIGURE 2.9 – Exemple d'annotations pour les instructions arithmétiques et logiques.

Les instructions de mouvement de données copient la valeur d'un opérande source vers l'un des douze registres généraux. Deux cas peuvent se produire : lorsque l'instruction de mouvement écrase complètement la donnée stockée dans le registre destination, et lorsque l'écrasement est partiel. En effet, les instructions peuvent permettre de copier uniquement la moitié de la valeur d'un registre vers le registre destination. Les instructions prennent la forme présentée par la Formule 2.9, avec OP le nom de l'instruction, Rd le registre destination et $Operand$ soit un registre source soit une valeur immédiate.

$$< OP >< Rd >< Operand > \tag{2.9}$$

Lors d'un mouvement sur une valeur complète, l'étiquette de l'opérande source vient écraser l'étiquette du registre destination.

$$\underline{Operand} \rightarrow \underline{Rd}$$

Au contraire, lors d'un mouvement partiel de données, l'étiquette précédente du registre destination doit être combinée avec celle de l'opérande source.

$$\underline{Rd} \underset{MOVE}{\oplus} \underline{Operand} \rightarrow \underline{Rd}$$

La Figure 2.10 présente des exemples d'instruction de mouvement des données et la propagation des étiquettes qu'elles engendrent.

```
mov r2, r3
movw  r0, #51795
```

$$\underline{r3} \rightarrow \underline{r2}$$
$$\underline{r0} \underset{MOVE}{\oplus} \underline{Imm} \rightarrow \underline{r0}$$

FIGURE 2.10 – Exemple d'annotations pour les instructions de mouvement de données.

Les instructions de comparaison permettent de comparer les valeurs des opérandes et de mettre à jour le registre APSR qui conditionnera l'exécution d'une instruction conditionnelle future. Ce type d'annotation nommé $COMPARE$, prend la forme présentée par la Formule 2.10, avec OP le nom de l'instruction, Rn le premier registre source et $Operand$ un deuxième registre source ou une valeur immédiate.

$$< OP >< Rn >< Operand > \tag{2.10}$$

Les flux d'information générés correspondent donc à la propagation des étiquettes des opérandes sources vers les champs du registre APSR.

$$\underline{Rn} \underset{COMPARE}{\oplus} \underline{Operand} \rightarrow \underline{APSR.N}$$
$$\underline{Rn} \underset{COMPARE}{\oplus} \underline{Operand} \rightarrow \underline{APSR.Z}$$
$$\underline{Rn} \underset{COMPARE}{\oplus} \underline{Operand} \rightarrow \underline{APSR.C}$$
$$\underline{Rn} \underset{COMPARE}{\oplus} \underline{Operand} \rightarrow \underline{APSR.V}$$

La Figure 2.11 présente un exemple d'instruction de comparaison de données et la propagation des étiquettes qu'elle engendre.

```
cmp r3, #6
```

$$\underline{r0} \underset{COMPARE}{\oplus} \underline{r1} \rightarrow \underline{APSR.V}$$
$$\underline{r3} \underset{COMPARE}{\oplus} \underline{Imm} \rightarrow \underline{APSR.N}$$
$$\underline{r3} \underset{COMPARE}{\oplus} \underline{Imm} \rightarrow \underline{APSR.Z}$$
$$\underline{r3} \underset{COMPARE}{\oplus} \underline{Imm} \rightarrow \underline{APSR.C}$$
$$\underline{r3} \underset{COMPARE}{\oplus} \underline{Imm} \rightarrow \underline{APSR.V}$$

FIGURE 2.11 – Exemple d'annotations pour les instructions de comparaison.

Les instructions arithmétiques et logiques utilisées pour mettre à jour les champs du registre *APSR*, de type $ARITH\_LOGIC\_AND\_COMPARE$. Les instructions arithmétiques et

logiques possèdent une version permettant d'affecter le résultat de leur opération aux champs du registre APSR. Pour cela, le suffixe « s » doit être ajouté au nom de l'instruction. Ces instructions ont la forme présentée par la Formule 2.11

$$< OP > s < Rd >< Rn >< Operand > \tag{2.11}$$

Ce type d'instruction génère un flux d'information entre les deux registres sources *Rn* et *Operand* vers le registre destination *Rd*. Comme le résultat de l'opération est stocké dans le registre *Rd*, on utilise l'étiquette de ce registre pour propager les étiquettes vers les champs du registre *APSR*.

$$\begin{aligned} \underline{Rn} \underset{ARITH\_LOGIC\_AND\_COMPARE}{\oplus} \underline{Operand} &\rightarrow \underline{Rd} \\ \underline{Rd} &\rightarrow \underline{APSR.N} \\ \underline{Rd} &\rightarrow \underline{APSR.Z} \\ \underline{Rd} &\rightarrow \underline{APSR.C} \\ \underline{Rd} &\rightarrow \underline{APSR.V} \end{aligned}$$

Les instructions de branchement génèrent des flux d'information lorsqu'ils sont conditionnels. En effet, lors d'un branchement conditionnel, les chemins empruntés dépendent de la valeur des champs du registre APSR utilisés par le code conditionnel. Ces instructions sont de type *BRANCH* et prennent la forme décrite dans la Formule 2.12, avec *Cond* le suffixe du code conditionnel présenté dans le Tableau 2.1.

$$< OP >< Cond >< Operand > \tag{2.12}$$

À partir de *Cond*, il est possible d'identifier le sous-ensemble des bits du registre APSR utilisé par le code conditionnel, que l'on note X. L'étiquette $\underline{X}$ correspond à la combinaison des étiquettes des registres de cet ensemble. Cette étiquette doit être elle-même combinée avec l'étiquette courante du registre PC, située en haut de la pile.

$$top_{\widehat{\underline{PC}}}() \underset{BRANCH}{\oplus} \underline{X}$$

L'étiquette résultant de ce calcul est ensuite utilisée pour mettre à jour la dernière étiquette de la pile ou empilée sur la pile. Pour pouvoir savoir

comment cette nouvelle étiquette doit être gérée sur la pile, une analyse statique est nécessaire, qui fait l'objet de la Section 2.4.4.

Nous avons présenté jusqu'à présent la propagation pour les instructions manipulant uniquement des registres. Nous présentons ensuite les annotations correspondant aux instructions réalisant des accès mémoire.

2.4.3 Annotations pour les instructions manipulant la mémoire

Les instructions de type $MEMORY\_INSTR$ permettent de réaliser des lectures et des écritures sur la mémoire. Ces accès mémoire induisent des flux d'information explicites et implicites. Les flux d'information implicites peuvent faire fuir de l'information et ainsi permettre à un attaquant d'en déduire une information secrète.

Prenons l'exemple d'un tableau contenant des valeurs publiquement connues, tel que des caractères ASCII utilisés par un programme pour chiffrer et déchiffrer des données à l'aide d'une substitution de caractères. Un attaquant capable d'observer les adresses utilisées lors des accès mémoires peut en déduire les indices du tableau utilisés pour la substitution, ce qui lui fournit ainsi une information sur la clef cryptographique utilisée.

Nous considérons que les flux d'information implicites peuvent générer de faux positifs et sont plus rares. C'est pourquoi dans cette thèse nous avons décidé de ne pas prendre en compte les flux implicites. Ainsi lors d'un accès mémoire, seuls l'étiquette de l'adresse mémoire et le registre impliqué dans l'opération sont utilisés.

$$\underline{address(value(Rs))} \rightarrow \underline{Rd}$$

Toutefois il serait possible d'étendre l'approche pour prendre en compte les flux implicites en prenant également en compte l'étiquette du registre stockant l'adresse mémoire, à savoir Rs.

$$\underline{Rs} \underset{MEMORY\_INSTR}{\oplus} \underline{address(value(Rs))} \rightarrow \underline{Rd}$$

Les instructions de type $MEMORY\_INSTR$ permettent de réaliser des accès mémoire en déterminant les adresses de plusieurs façons que l'on appelle « mode d'adressage ». L'adressage indirect permet d'utiliser la valeur d'un ou plusieurs registres comme adresse d'accès. Dans l'architecture ARM V7, il existe six manières différentes d'accéder à la mémoire de façon indirecte :

Registre de base : adressage indirect en utilisant la valeur du registre Rs comme adresse. Exemple :

```
ldr Rd, [Rs]
```

Ce mode d'adressage génère l'annotation suivante :

$$\underline{address(value(Rs))} \rightarrow \underline{Rd}$$

Registre de base avec déplacement (Pre-indexed), adressage indirect en utilisant la valeur de Rs additionné à une valeur immédiate.

Exemple :

```
ldr Rd, [Rs, Imm]
```

Ce mode d'adressage génère l'annotation suivante :

$$\underline{address(value(Rs) + Imm)} \rightarrow \underline{Rd}$$

Registre de base avec déplacement puis modification (Pre-indexed), adressage indirect en utilisant la valeur de Rs additionné à une valeur immédiate comme adresse puis écriture du résultat dans le registre Rs.

Exemple :

```
ldr Rd, [Rs, Imm]!
```

Ce mode d'adressage génère les annotations suivantes :

$$\underline{address(value(Rs) + Imm)} \rightarrow \underline{Rd}$$

Registre de base puis modification (Post-indexed), Adressage indirect en utilisant la valeur de Rs comme adresse puis incrémentation du registre Rs par l'immédiat.

Exemple :

```
ldr Rd, [Rs], Imm
```

Ce mode d'adressage génère l'annotation suivante :

$$\underline{address(value(Rs))} \rightarrow \underline{Rd}$$

Double registre de base , adressage indirect en utilisant l'addition des valeurs de Rs et Ra comme adresse Exemple :

```
ldr Rd [Rs, Ra]
```

Ce mode d'adressage génère l'annotation suivante :

$$\underline{address(value(Rs) + value(Ra))} \rightarrow \underline{Rd}$$

Double registre de base avec décalage , adressage indirect en utilisant l'addition des valeurs de Rs et Ra puis un décalage arithmétique ou logique (par exemple la fonction LSL qui permet de réaliser un décalage logique à gauche de Imm bits) comme adresse.

```
ldr Rd [Rs, Ra, LSL Imm]
```

Ce mode d'adressage génère l'annotation suivante :

$$\underline{address(LSL((value(Rs) + value(Ra)), Imm))} \rightarrow \underline{Rd}$$

L'adressage direct est impossible à réaliser pour les instructions mémoires des architectures ARMv7. En effet, les adresses sont représentées sur 32 bits et les instructions sont également représentées sur 32 bits. Il est donc impossible d'encoder une adresse complète dans une instruction. Le compilateur est donc chargé de traduire le mode d'adressage direct pour les instructions mémoires vers un adressage indirect. L'assembleur ARM V7 permet d'exprimer des accès directs, comme dans le code suivant :

```
ldr Rd, =0x52914320
```

Il s'agit en réalité d'une pseudo-instruction qui sera traduite par une suite de trois instructions. Les deux premières servent à écrire les 16 premiers bits (`movt`) puis les 16 bits de poids faible (`movw`) dans le registre Rs pour qu'il puisse contenir la valeur 0x52914320. La dernière instruction permet de charger les données présentent en mémoire à l'adresse spécifiée par le registre Rs et de les stocker dans le registre Rd.

```
movt Rs, 0x5291
movw Rs, 0x4320
ldr  Rd, [Rs]
```

Les instructions de type $MEMORY\_INSTR$ dépendent donc de la valeur des registres sources qui sont utilisés pour définir l'adresse mémoire accédée. Or ces valeurs ne sont pas connues du moniteur de sécurité. C'est pourquoi une instrumentation du code de l'application doit être menée avant certaines instructions d'accès mémoire afin d'informer le moniteur de sécurité de la valeur de ces registres. Cette instrumentation est décrite en section 2.5.

2.4.4 Gestion des flux implicites à l'aide de la pile d'étiquettes du registre PC

Pour pouvoir prendre en compte les flux d'information implicites lors de l'exécution du programme, il est nécessaire de propager les étiquettes des champs du registre APSR vers celle du registre PC lors d'un branchement conditionnel, mais également de propager l'étiquette du registre PC vers les instructions exécutées dans un bloc de base dépendant de ladite condition. Ainsi chaque annotation A de ce bloc de base, exécutée par le moniteur de sécurité, est étendue avec l'opération suivante.

$$A \underset{BRANCH}{\oplus} top_{\widehat{\underline{PC}}}()$$

Il est toutefois primordial de délimiter l'étendue de l'influence du branchement conditionnel. Il doit donc y avoir non pas une étiquette du registre PC, mais une multitude d'étiquettes correspondant aux différents blocs de base conditionnels qui ont été traversées lors de l'exécution du programme. Pour pouvoir connaître les opérations à mener pour la gestion de la pile d'étiquette du registre PC, une analyse statique présentée par l'algorithme 3 est nécessaire. Lorsqu'un bloc de base BB contient un branchement conditionnel, son post-dominateur immédiat $IDomBB$ délimite la fin de l'influence qu'a le code conditionnel sur les instructions exécutées. Il est donc nécessaire d'empiler une nouvelle étiquette avant l'instruction de saut conditionnel présente dans le bloc de base BB et de la désempiler à la fin du bloc de base $IDomBB$.

Certains cas particuliers doivent tout de même être pris en compte. En effet, les blocs de base constituant une boucle ne doivent pas empiler les étiquettes mais les mettre à jour. Pour cela, la création d'un *pré-header* est parfois nécessaire afin d'empiler l'étiquette du registre PC avant l'entrée dans la boucle. Il est également nécessaire d'équilibrer le nombre d'empilements et de dépilements lorsqu'un même bloc de base représente le post-dominateur immédiat de plusieurs blocs de bases contenant des branchements conditionnels.

La Figure 2.12 présente les transformations et les opérations de gestion de la pile réalisées par l'algorithme 3. L'opération $PUSH$ permet d'empiler une étiquette sur la pile, $UPDATE$ permet de mettre à jour l'étiquette présente sur le haut de la pile et POP permet de désempiler une étiquette. Les blocs de base violets sont des pré-headers créés pour pouvoir empiler en avant l'entrée de la boucle, les blocs de base jaunes sont à l'intérieur d'une boucle, les blocs de base noirs sont créés afin de désempiler une étiquette si l'immédiat post-dominateur fait partie d'une boucle, les blocs de base bleus correspondent

Algorithme 3 : Analyse statique permettant la mise à jour de l'étiquette du registre PC.

```
pour chaque Fonction F faire
    pour chaque Bloc de base BB faire
        si BB possède un branchement conditionnel alors
            IDomBB = le postdominateur immédiat de BB
            si BB est dans une boucle alors
                si BB est le point d'entrée de la boucle alors
                    Créer un bloc de base H avant BB et y ajouter
                     une annotation PUSH_PC
                    si IDomBB est dans une autre boucle alors
                        Créer un bloc de base H avant IDomBB et y
                         ajouter une annotation POP_PC
                    sinon
                        Ajouter une annotation POP_PC dans
                         IDomBB
                sinon
                    Ajouter une annotation UPDATE_PC dans BB
            sinon
                Ajouter une annotation PUSH_PC à la fin de BB
                Ajouter une annotation POP_PC au début de
                 IDomBB
                 = tous les blocs de base accessibles entre BB et
                 IDomBB
                pour chaque Bloc de base S ⊂ 𝔸 ∪ {IDomBB} faire
                    pour chaque Bloc de base P antécédent du bloc de
                     base S faire
                        si P ∉ 𝔸 ∪ {BB, IDomBB} alors
                            Créer un bloc de base entre P et S et y
                             ajouter une annotation PUSH_PC
                    fin
                fin
    fin
fin
```

à des immédiats post-dominateurs qui permettent d'insérer simplement une opération de dépilement. Pour finir, les blocs de base rouges sont les blocs créés pour équilibrer le nombre d'empilements des étiquettes selon le nombre de dépilements de leur post-dominateur immédiat commun.

Soit $\underline{X}$ la combinaison des étiquettes des champs du registre APSR utilisés

pour un branchement conditionnel. Ainsi, cette analyse statique permettra de générer l'annotation suivante dans les blocs de base violets, à savoir un simple empilement de l'étiquette au sommet de la pile :

$$push_{\underline{\widehat{PC}}}(pop_{\underline{\widehat{PC}}})$$

Les blocs de base jaunes devront eux mettre à jour l'étiquette au somme de la pile à l'aide de l'annotation suivante :

$$push_{\underline{\widehat{PC}}}(top_{\underline{\widehat{PC}}} \underset{BRANCH}{\oplus} \underline{X})$$

Les blocs de base noirs et bleus devront désempiler une étiquette à l'aide de l'annotation suivante :

$$pop_{\underline{\widehat{PC}}}()$$

Les blocs de base verts et rouges devront empiler une nouvelle étiquette à l'aide de l'annotation suivante :

$$push_{\underline{\widehat{PC}}}(top_{\underline{\widehat{PC}}} \underset{BRANCH}{\oplus} \underline{X})$$

Après avoir décrit les différentes annotations générées par le compilateur pour refléter les flux d'informations liées à l'exécution des instructions de chaque bloc de base, nous décrivons dans la section suivante la phase d'instrumentation, qui permet d'envoyer au moniteur les adresses de certains accès mémoire.

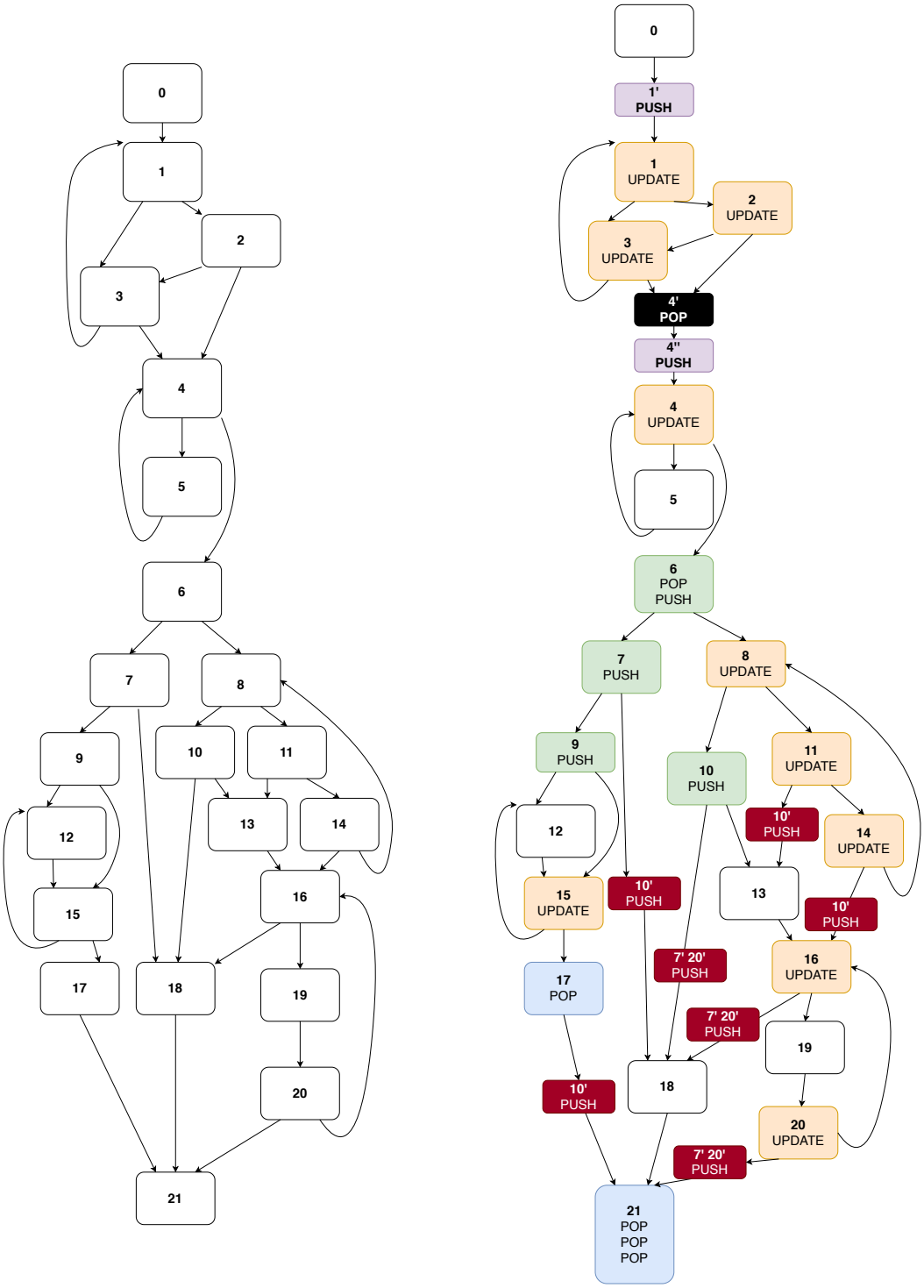


FIGURE 2.12 – Résultat de l'analyse statique pour gérer les flux implicites.

2.5 Extraction d'informations par instrumentation du programme utilisateur

L'instrumentation de l'application doit permettre d'envoyer des données vers le moniteur de sécurité à travers le bus AXI en utilisant des FIFO. Puisque l'application utilisateur n'est pas privilégiée, il n'est pas possible pour l'application de communiquer directement avec le moniteur de sécurité.

Deux approches permettent l'instrumentation de l'application, la première est la création ou l'invocation d'un appel système pour pouvoir envoyer les données, en déroutant le flot de contrôle de l'application et en changeant le niveau de privilège du processeur. Cette méthode utilise un pilote Linux dans le noyau. La deuxième approche est de faire correspondre l'adresse physique de la FIFO à une adresse virtuelle de l'application, lors de son chargement, à l'aide d'un pilote Userspace I/O (UIO) [78]. Cette adresse pourra ainsi être directement utilisée par le code de l'application pour envoyer des données vers le moniteur de sécurité sans nécessiter de réaliser un appel système pour chaque écriture vers la FIFO. La deuxième approche a été choisie, car elle permet une instrumentation beaucoup plus simple. En effet, une seule instruction ARM est nécessaire pour envoyer une donnée, et aucun flot de contrôle supplémentaire n'est ajouté.

On ajoute dans l'application utilisateur, un symbole externe qui sera défini lors du chargement de l'application. Ce symbole contient l'adresse virtuelle d'une FIFO du moniteur de sécurité. Lors du chargement de l'application, le pilote UIO est appelé permettant d'associer une adresse virtuelle à cette FIFO. Puisque l'architecture ARM ne permet pas l'accès direct à la mémoire, un registre doit être utilisé pour pouvoir stocker cette adresse. Nous avons décidé de réserver le registre *r9* à cet usage. Le *back-end* du compilateur a été modifié afin qu'il n'utilise pas ce registre lors de la compilation du code de l'application. Le code 2.4 illustre cette approche.

```
0  /* On ouvre le fichier correspondant au pilote UIO */
1  uio_hardblare_instr_fd = open("/dev/uio0", O_RDWR);
2
3  ...
4
5  /* On recupere une adresse virtuelle correspondant a la FIFO du moniteur */
6  __hardblare_instr_addr = mmap( .., uio_hardblare_instr_fd);
7
8  ...
9
10 /* On sauvegarde l'adresse dans le registre r9 */
11 __asm volatile("mov r9, %[input]"
12                 : // No output
```

```
13                  :[input] "r" (__hardblare_instr_addr)
14                  );
```

CODE SOURCE 2.4 – Initialisation de l'instrumentation

Une fois cette initialisation terminée, une seule instruction est nécessaire pour envoyer des données au moniteur. Supposons que le programme utilisateur accède à une adresse mémoire stockée dans le registre *r0* et supposons que cette valeur ait été fournie par un appel système ou calculée par l'application à l'aide d'opérations arithmétiques. Nous avons ainsi l'instruction suivante :

```
ldr r5, [r0]
```

Puisque le moniteur ne connaît pas la valeur du registre *r0*, on va alors l'envoyer au moniteur de sécurité à l'aide de l'instruction suivante :

```
str r0, [r9]
```

Cette instruction est insérée avant chaque instruction accédant à la mémoire, de type *MEMORY_INSTR*, dont la valeur du registre de base n'est pas connue par le moniteur.

Lors de l'envoi de données au moniteur de référence, il est important que le moniteur se mette en attente d'une valeur de la part de l'application utilisateur. Pour cela, une annotation est générée pour permettre au moniteur de se mettre en attente d'une valeur émise via la FIFO d'instrumentation, de la lire lorsque la valeur est présente et de la stocker dans un registre pour être utilisé lors de la propagation d'étiquettes vers ou depuis la mémoire.

L'instrumentation d'une application implique un surcoût en termes de performance puisque l'écriture d'une donnée vers le moniteur est effectuée. Par conséquent, l'instrumentation doit être utilisée avec parcimonie. De plus, puisque les variables du programme sont stockées dans la pile, la grande majorité des instructions accédant à la mémoire utilisent les registres SP et Frame Pointer (FP). En effet, le registre SP fait référence au haut de la pile tandis que le registre FP désigne l'adresse du cadre de pile. Par conséquent, la première optimisation qui peut être réalisée, afin de limiter le nombre d'instrumentations, consiste à suivre toutes les modifications des registres SP et FP dans l'application et de retranscrire ces modifications dans le moniteur de sécurité. Ainsi, lors d'accès mémoire sur la pile, aucune instrumentation n'est alors nécessaire, permettant de diminuer le surcoût lié à l'envoi de données vers le moniteur de sécurité.

Dans le but d'évaluer les bénéfices apportés par cette optimisation, nous

avons utilisé un ensemble d'applications provenant du *benchmark* MiBench [3]. La Figure 2.13 présente les différents surcoûts en terme de temps d'exécution de l'instrumentation selon l'approche naïve et l'approche optimisée qui maintient une copie de SP, comparé à la version originale de l'application (non instrumentée).

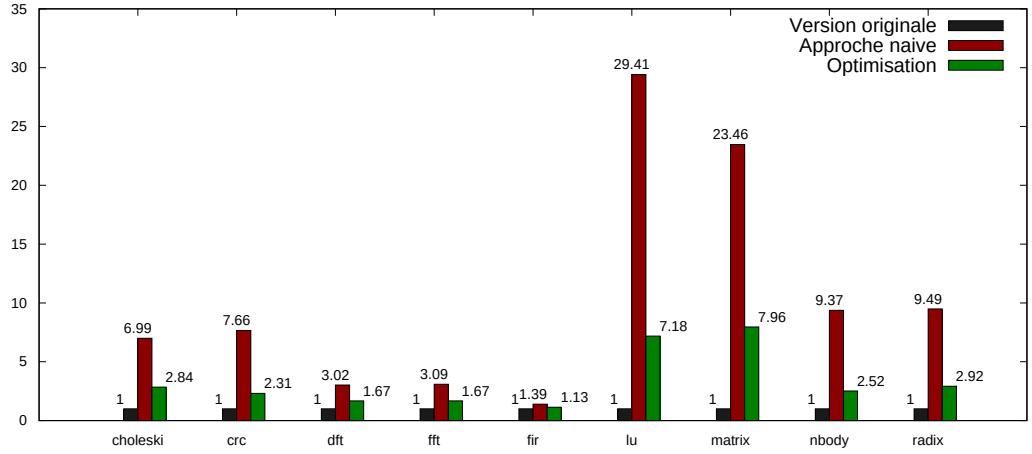


FIGURE 2.13 – Impact de l'instrumentation sur le temps d'exécution.

Le surcoût en terme de temps d'exécution est en moyenne de 24,6% pour l'approche naïve, lorsque l'optimisation permet de descendre à un surcoût moyen de 5,37%. La taille du fichier exécution est également affectée par l'instrumentation, comme l'illustre la Figure 2.14. La taille varie ainsi en moyenne de 10% pour l'approche naïve, lorsque l'optimisation permet de descendre à un surcoût moyen de 3%.

Parfois, plusieurs valeurs doivent être envoyées au moniteur de sécurité. Ces envois peuvent être regroupés et envoyés avec seulement une instruction. La version multiple de l'instruction *str*, à savoir l'instruction *stm*, permet d'envoyer la valeur de plusieurs registres en même temps. L'instruction *stm* possède trois versions. La première que l'on nommera stm_2 permet d'envoyer deux données, stm_3 permet d'envoyer trois données et stm_4 pour envoyer quatre données. Par exemple, lorsque l'instruction utilise deux registres pour calculer l'adresse :

```
ldr r5 [r0, r1]
```

On instrumente le code en envoyant les valeurs de ces deux registres en même temps avec l'instrumentation suivante :

```
stm r9, {r0, r1}
```

3. http://vhosts.eecs.umich.edu/mibench/

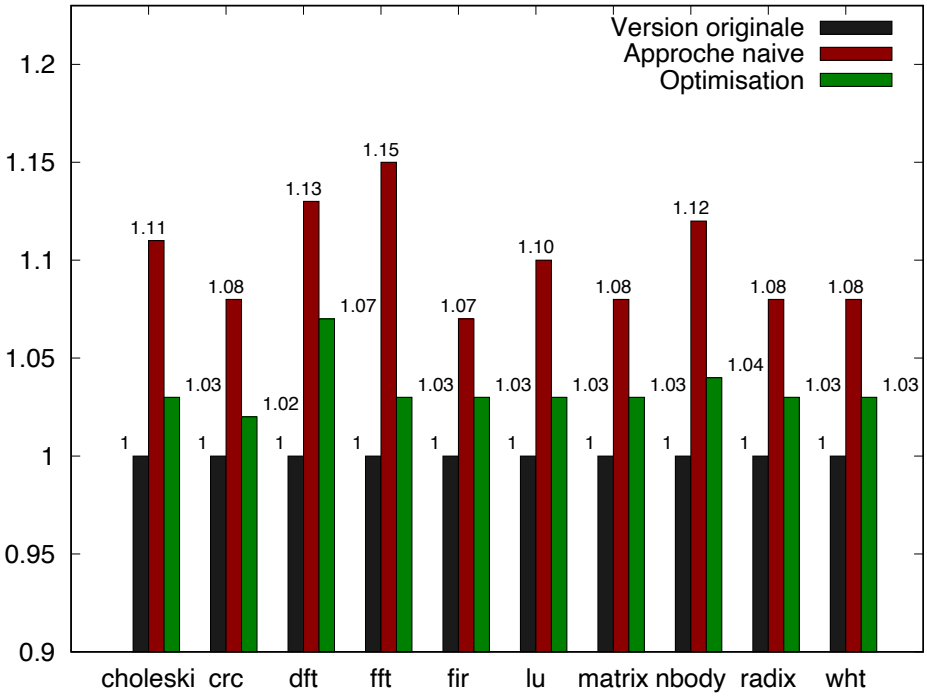


FIGURE 2.14 – Impact sur la taille des fichiers exécutables.

L'utilisation de l'instruction *stm* permet de réduire le surcoût qu'apporte l'instrumentation lorsque plusieurs envois doivent être réalisés.

Cette optimisation consiste en une analyse intraprocédurale permettant d'instrumenter le code seulement au moment opportun. Chaque instruction nécessitant une instrumentation est analysée. Si les registres contenant les informations à envoyer au moniteur n'ont pas été redéfinis depuis l'instruction précédente nécessitant une instrumentation, on va alors retarder au maximum l'instrumentation. L'algorithme 4 présente l'analyse statique permettant ainsi d'optimiser l'instrumentation.

La Figure 2.15 montre le gain en millisecondes que met l'application pour envoyer des adresses au coprocesseur via le mécanisme d'instrumentation, en fonction du nombre d'accès mémoire. On peut remarquer un gain allant de 40% à 50% en moyenne lorsque plusieurs adresses peuvent être envoyées simultanément à l'aide de l'instruction `stm`.

Dans cette section, nous avons décrit la phase d'instrumentation permettant d'envoyer au coprocesseur les adresses des accès mémoire qui ne peuvent être prédites statiquement. Cette instrumentation repose sur la réservation d'un registre permettant de stocker l'adresse virtuelle de la FIFO, allouée

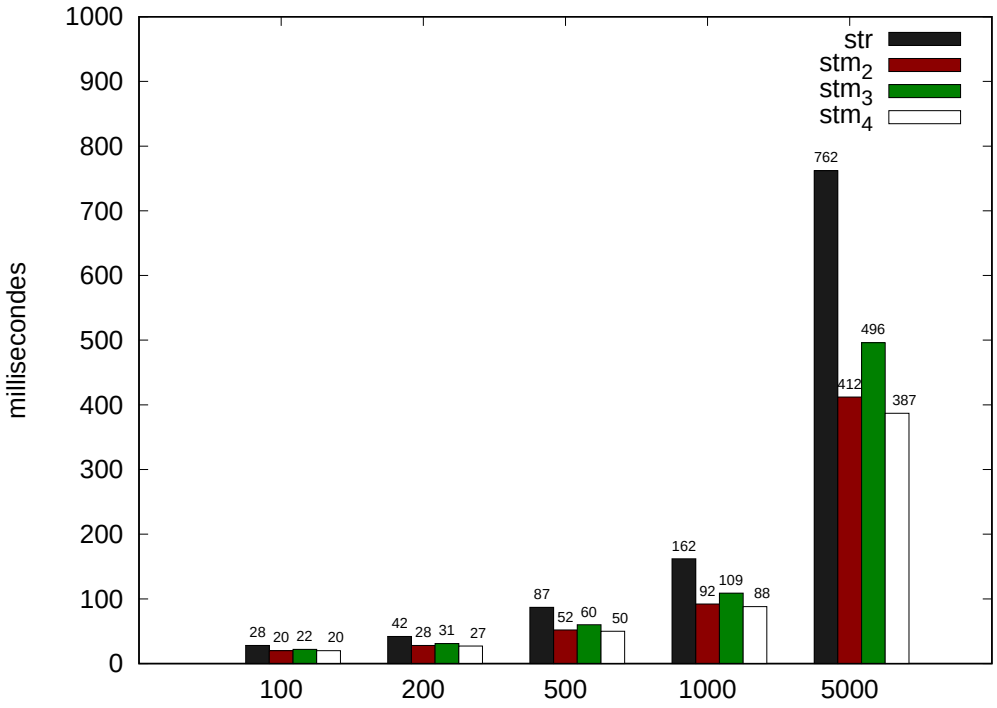


FIGURE 2.15 – Optimisation de l'instrumentation lors d'envoi multiple.

à l'aide d'un pilote UIO. J'ai par ailleurs développé une phase d'optimisation permettant d'envoyer plusieurs adresses simultanément, ce qui limite le surcoût de l'instrumentation. La section suivante décrit les échanges entre le coprocesseurs et l'OS.

Algorithme 4 : Optimisation de l'instrumentation.

```
pour chaque Fonction F faire
    pour chaque Bloc de base BB faire
        VInstr = ∅
        pour chaque Instruction I faire
            si I nécessite une instrumentation alors
                pour chaque I' dans VInstr faire
                    si les opérandes de I à envoyer sont redéfinies
                     entre I et I' alors
                        NumeroPosition = position de I' dans VInstr
                        Ajouter une instruction d'instrumentation
                         stm_NumeroPosition après l'instruction I'
                        Supprimer les NumeroPosition instructions
                         dans VInstr
                    fin
                fin
                Ajouter I à VInstr
                si taille(VInstr) == 4 alors
                    Ajouter une instruction d'instrumentation stm_4
                     après l'instruction I
                    Supprimer les quatre instructions dans VInstr
                fin
            fin
        fin
        Ajouter une instruction d'instrumentation stm_taille(VInstr)
         après la dernière instruction dans VInstr
        Supprimer les instructions stockées dans VInstr
    fin
fin
```

2.6 Analyse dynamique lors de la communication avec le système d'exploitation

Les applications utilisateurs s'exécutent sans privilège sur le système. L'accès à certaines ressources leur est donc limité et les applications doivent obligatoirement passer par le système d'exploitation pour utiliser les services et ressources mis à disposition par ce dernier.

Prenons l'exemple du code 2.5, lorsque l'application demande une projection mémoire d'un fichier, une zone mémoire est réservée pour y projeter les données du fichier. La ligne 25 appelle la fonction `mmap` et stocke l'adresse virtuelle retournée dans la variable `mmaped`. Cette adresse retournée par l'appel système ne peut être prédite avant l'exécution car elle dépend de plusieurs facteurs.

```
0
1  int main ()
2  {
3      /* The file descriptor. */
4      int fd;
5      /* Information about the file. */
6      struct stat s;
7      int status;
8      size_t size;
9      /* The file name to open. */
10     const char * file_name = "test.txt";
11     /* The memory-mapped thing itself. */
12     const char * mapped;
13     int i;
14
15     /* Open the file for reading. */
16     fd = open ("test.txt", O_RDONLY);
17     check (fd < 0, "open %s failed: %s", file_name, strerror (errno));
18
19     /* Get the size of the file. */
20     status = fstat (fd, & s);
21     check (status < 0, "stat %s failed: %s", file_name, strerror (errno));
22     size = s.st_size;
23
24     /* Memory-map the file. */
25     mapped = mmap (0, size, PROT_READ, MAP_PRIVATE, fd, 0);
26     check (mapped == MAP_FAILED, "mmap %s failed: %s",
27            file_name, strerror (errno));
28
29     /* Now do something with the information. */
30     for (i = 0; i < size; i++) {
31         char c;
32         c = mapped[i];
```

```
33                  ...
34      }
35
36      return 0;
37  }
```

CODE SOURCE 2.5 – Exemple d'une projection mémoire d'un fichier

Après la compilation de ce programme, on obtient les instructions machine suivantes :

```
000117b4 <mmap>:
...
11840: ef000000   svc 0x00000000
...

00010328 <main>:
...
104d0: bl 117b4 <mmap>
104d4: ldr  r5, [r0, r1]
...
```

L'exécution de la fonction principale (*main*) de l'application appelle la fonction *mmap* qui provoque un appel système permettant de projeter en mémoire le fichier *test.txt* et de retourner l'adresse virtuelle de cette projection dans le registre *r0* conformément à l'Application Binary Interface (ABI). Cela génère un flux d'information du fichier vers la zone mémoire projetée. Lorsque le fichier *test.txt* est étiqueté avec un niveau de sécurité, cette étiquette doit être propagée vers la zone mémoire concernée. Puisque le système d'exploitation est responsable de ce flux, il doit en informer le moniteur de sécurité.

Ce registre *r0* est ensuite utilisé pour accéder aux données du fichier. L'adresse de l'instruction de chargement est calculée à l'aide des valeurs du registre *r0* contenant l'adresse virtuelle de base de la projection, et du registre *r1* contenant la valeur de la variable *i*. Puisque cette adresse ne peut pas être connue statiquement, il est nécessaire d'en informer le moniteur en instrumentant le code, comme présentée en section 2.5.

Lors d'un appel système, des flux d'information peuvent se produire dans le système.

Afin de prendre en compte les flux d'informations mettant en jeu des fichiers comme conteneurs d'informations, il est nécessaire :

1. De pouvoir associer une étiquette à ces conteneurs, et ce de manière

persistante (les fichiers étant des conteneurs persistants, contrairement à la mémoire et aux registres) ;

2. De pouvoir identifier les flux d'informations entre les fichiers et les autres conteneurs de l'application, afin de propager les étiquettes de ces conteneurs en conséquence.

2.6.1 Étiquetage des fichiers

La plupart des systèmes de fichiers utilisés par Linux supportent les attributs étendus qui offrent un mécanisme pour pouvoir stocker des paires de clé-valeur. Ces attributs permettent ainsi de stocker de façon persistante des métadonnées pour chaque fichier du système. La sémantique de ces métadonnées est fixée par l'utilisateur. Dans notre cas, deux attributs étendus sont nécessaires pour supporter le contrôle de flux d'information

Le premier attribut étendu, appelé *iTag*, nous permet de stocker une étiquette représentée sur 32 bits et représentant le niveau de sécurité du fichier. Ainsi, chaque fichier a la possibilité d'être étiqueté avec un niveau de sécurité. Lors de la manipulation de fichiers produisant un flux d'information, on utilise cet attribut étendu pour propager l'étiquette dans le système.

Le deuxième attribut étendu appelé *pTag*, contient quant à lui un ensemble de niveaux de sécurité acceptés par le fichier. Lorsqu'une application souhaite écrire des données dans un fichier, on vérifie que le niveau de sécurité des données à écrire est inclus dans l'ensemble de niveaux *pTag*.

2.6.2 Suivi de flux d'information mettant en jeu des fichiers

Le développement d'un support pour une fonctionnalité de sécurité comme le DIFT dans un système d'exploitation moderne pose un certain nombre de problèmes d'implémentation. En effet, les différentes couches d'abstraction, le contexte multitâche et préemptif, les accès concurrents aux ressources du système comme la mémoire partagée rendent difficile l'implémentation de fonctionnalités de sécurité.

La NSA avait besoin d'un système d'un système d'exploitation capable de gérer des politiques de contrôle d'accès multiniveau et a donc proposé SELinux, une version modifiée du noyau Linux. La communauté Linux a refusé ces modifications dans le noyau, car elle souhaitait une solution générique et modulable à l'image des pilotes, pour pouvoir implémenter des fonctionnalités de sécurité facilement sous forme de modules.

La communauté Linux a ainsi proposé les Linux Security Modules, une interface permettant d'ajouter des modules de sécurité. Cette interface est basée sur les crochets LSM : des appels vers des pointeurs de fonctions à définir par le développeur, qui sont placés à des points clefs avant chaque accès à une ressource du système. Ces crochets ne sont pas nécessairement placés dans le code des appels système, ils sont également présents dans des fonctions internes au noyau. Les crochets LSM ont été pensés avec la finalité d'implémenter un contrôle d'accès supplémentaire à celui existant dans le noyau. L'utilisation de ces crochets LSM nous a donc permis de développer un module de sécurité pour le contrôle de flux d'information capable de communiquer avec le moniteur de sécurité.

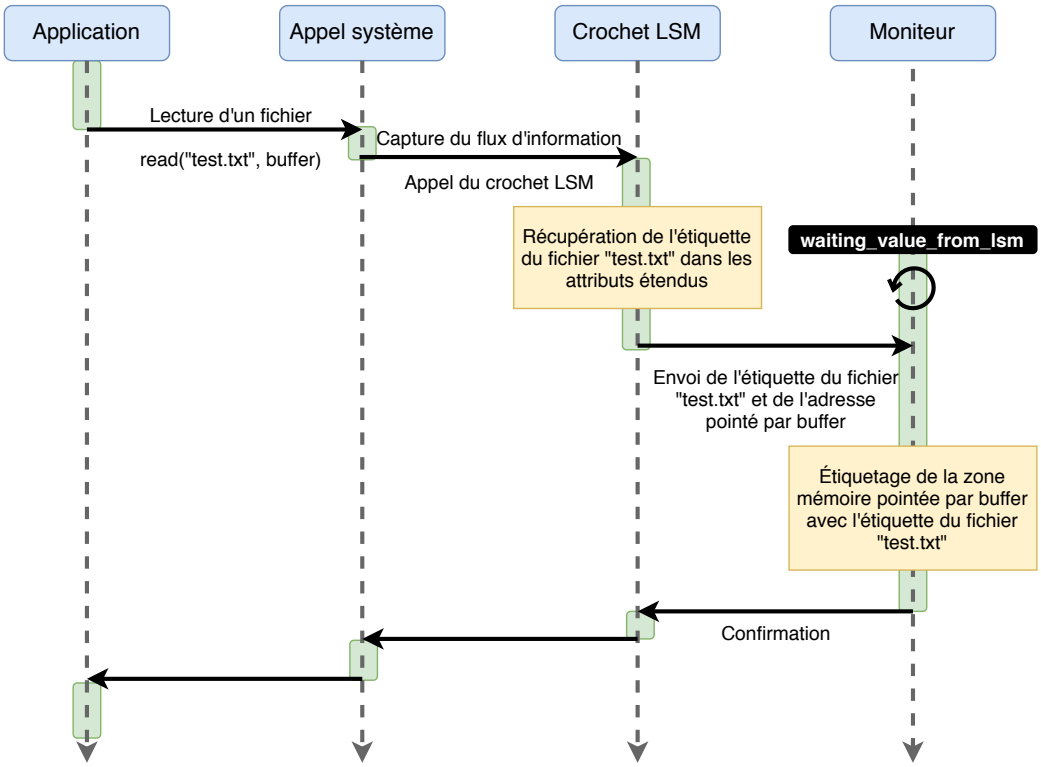


FIGURE 2.16 – Protocole de communication entre le système d'exploitation et le moniteur lors de la lecture d'un fichier

La Figure 2.16 illustre le protocole qui se produit lors d'un appel système permettant la lecture d'un fichier. Lors de la lecture d'un fichier, l'application réalise un appel système, matérialisé par l'exécution de l'instruction *xvc* dans l'architecture ARM V7 [4]. Lors de la phase d'analyse statique du code de

4. Cette instruction est généralement présente dans le code d'une bibliothèque utilisée

l'application, nous produisons une annotation *waiting_value_from_lsm* à la fin des annotations correspondant au bloc de base dans lequel est contenu l'instruction *svc* comme précisé en section 2.4.2. Ainsi, lors de l'exécution de l'application, le moniteur DIFT se met en attente d'un message de l'OS lorsqu'un appel système est réalisé par l'application. Lors du traitement de l'appel système par le code du noyau, un crochet LSM est pris, ce qui permet de propager les étiquettes entre le système de fichiers et le moniteur de sécurité. Plus précisément, lors de la lecture du fichier vers une zone tampon, l'étiquette stockée dans les attributs étendus du fichier est récupérée, puis transmise au moniteur de sécurité en spécifiant l'étiquette, l'adresse et la taille de la zone tampon. Le moniteur peut alors associer l'étiquette à l'ensemble de la zone mémoire correspondant au tampon, puis propager cette étiquette lorsque le code de l'application recopie les données de ce tampon dans d'autres conteneurs.

Lors de l'écriture dans un fichier, illustrée par la Figure 2.17, une requête est réalisée au moniteur de sécurité afin de récupérer l'étiquette de la zone mémoire à écrire dans le fichier. Le moniteur envoie donc l'étiquette de la zone mémoire qui sera ensuite ajoutée aux attributs étendus du fichier.

Afin d'implémenter cette approche, j'ai modifié le code de RfBlare [5], un moniteur réalisant du DIFT à gros grain au niveau noyau, afin de désactiver la propagation des étiquettes au sein du module de sécurité LSM de RfBlare.

RfBlare permet de suivre tous les flux d'informations qui se produisent au niveau du système d'exploitation. En effet, il a été formellement prouvé que tous les flux qui se produisent dans RfBlare passent systématiquement par un crochet LSM. On a ainsi une couverture complète sur les différents appels système manipulant les fichiers.

J'ai par la suite implémenté les protocoles décrits précédemment, à la fois côté noyau et dans le *firmware* du moniteur DIFT. Cela permet ainsi de déléguer cette tâche au moniteur de sécurité.

2.7 Stockage des annotations

Une fois les annotations générées lors de la compilation de l'application, il est nécessaire de les stocker pour pouvoir par la suite les transmettre au coprocesseur lors de l'exécution de l'application. Nous avons fait le choix de stocker ces annotations au sein même du fichier exécutable de l'application. Les annotations doivent être stockées selon l'adresse des blocs de base aux-

par l'application. Le code des bibliothèques utilisées par l'application suivie par le moniteur doit donc avoir été annoté et instrumenté.

5. http://www.blare-ids.org/rfblare/

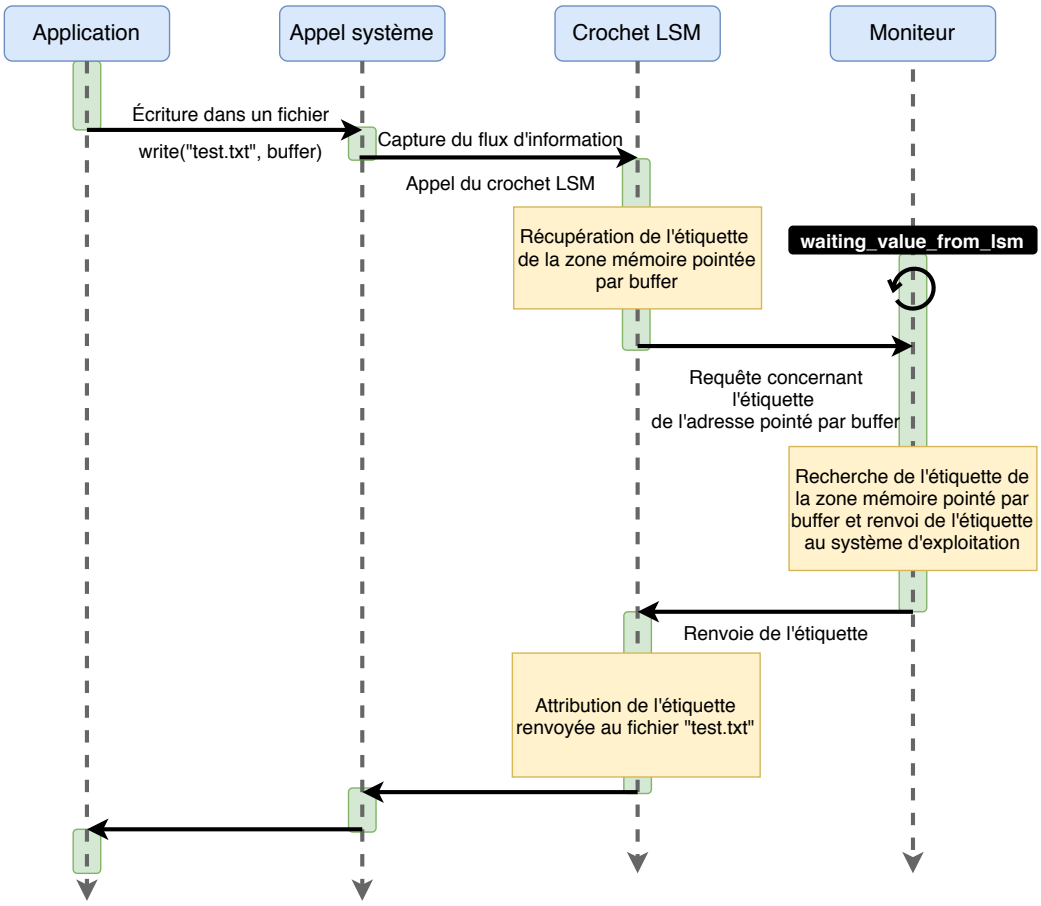


FIGURE 2.17 – Protocole de communication entre le système d'exploitation et le moniteur lors de l'écriture d'un fichier

quels elles correspondent afin d'être retrouvées facilement lors de la récupération des traces du PTM par le coprocesseur. Sous Linux, le format de fichier exécutable par défaut est le format ELF. Un fichier ELF est décomposé en segments qui possèdent chacun un rôle bien précis et sont eux-mêmes composés de sections. Nous avons ainsi créé deux segments réservés uniquement au stockage des annotations.

La Figure 2.18 illustre le stockage des annotations dans le fichier exécutable. Le segment *LOAD* est chargé dans la mémoire de l'application lors de son exécution et comporte notamment la section *.text* qui contient le code machine de l'application. Nous avons ajouté deux segments pour stocker les annotations. Ces segments ne sont pas chargés dans l'espace mémoire de l'application mais permettent au chargeur de programme de l'OS de les envoyer au coprocesseur. Le premier de ces segments, noté *HARDBLARE_BBT*,

contient la section *.HB.bbt* qui est composée d'un tableau permettant de faire le lien entre une adresse du code de l'application et l'adresse où se trouvent ces annotations. On appelle ce tableau Basic Block Table (BBT). Le second segment, appelé *HARDBLARE_ANNOT*, inclut la section *.HB.annot* qui contient les annotations à proprement parler.

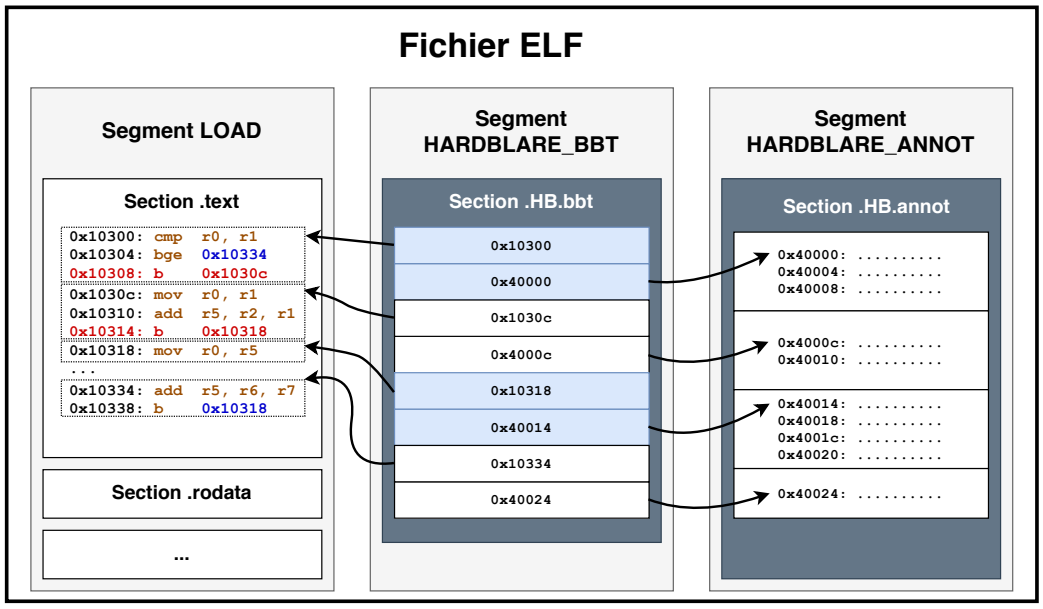


FIGURE 2.18 – Stockage des annotations dans un fichier exécutable ELF

Pour pouvoir remplir le BBT, il est nécessaire, lors de la compilation et de l'édition de liens avec des bibliothèques externes, de calculer les adresses finales de chaque bloc de base et l'adresse des annotations correspondantes dans la section *.HB.annot*. Il s'agit d'un calcul de *relocation*. Ce calcul des adresses finales a nécessité la modification de l'éditeur de lien « GNU Linker » afin de calculer des adresses relatives au début du segment *HARDBLARE_ANNOT* au lieu d'avoir une adresse absolue. En effet, le moniteur de sécurité stocke ces sections dans une mémoire qui lui est dédiée, il est donc nécessaire d'avoir des adresses relatives au segment.

Lors du lancement d'une application dans le système d'exploitation, le chargeur du noyau est chargé d'analyser la validité du fichier exécutable et de charger en mémoire les différents segments le constituant. Les segments *HARDBLARE_BBT* et *HARDBLARE_ANNOT* sont ainsi chargés dans une mémoire dédiée du moniteur de sécurité, comme illustré par la figure 2.19. Pour chaque trace PTM reçue correspondante à l'adresse d'un bloc de base, le moniteur de sécurité va parcourir la section *HARDBLARE_BBT* à la recherche de l'emplacement des annotations pour ce bloc de base.

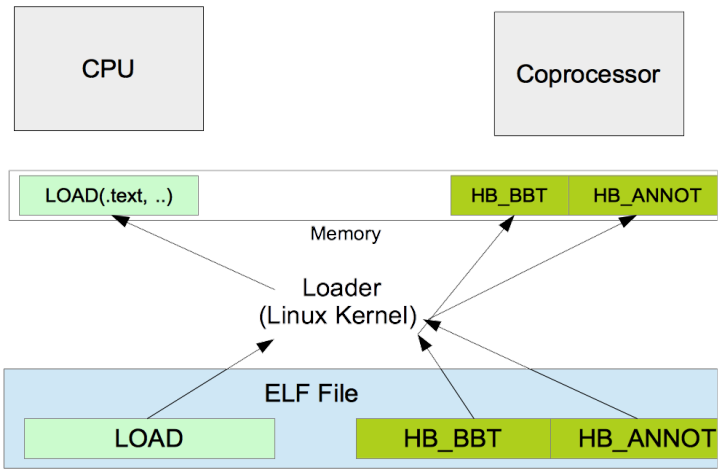


FIGURE 2.19 – Chargement d'un fichier exécutable ELF

Dans cette section, nous avons décrit notre approche permettant le stockage des annotations du programme utilisateur dans des segments et sections ELF spécifiques de l'exécutable, en prenant en compte de la résolution des adresses des blocs de base lors de l'édition de liens, ainsi que le chargement de ces segments dans la mémoire du moniteur de sécurité.

La génération et le chargement des annotations permettent, avec les informations reçues via les composants de débogage et de traces, de connaître le chemin emprunté par l'application utilisateur dans le graphe de flot de contrôle et de propager les étiquettes. Ces informations ne sont malheureusement pas suffisantes, car les données générées dynamiquement lors de l'exécution de l'application ne peuvent être inférées statiquement et ne sont pas connues du moniteur. Il est donc nécessaire d'envoyer ces informations connues uniquement lors de l'exécution, par un canal de communication utilisant le bus AXI et menant au moniteur. Une analyse est donc nécessaire pour instrumenter l'application et envoyer de façon dynamique ces informations. Certains appels système modifient également le comportement ou la mémoire de l'application utilisateur. Il est donc également nécessaire lors d'un appel système, d'informer le moniteur qu'une zone mémoire a par exemple été allouée ou libérée, ou qu'un flux d'information s'est produit au niveau des structures de données du processus. Ces différents échanges ont nécessité d'adapter le cœur du moniteur de sécurité, que nous présentons dans la section suivante.

2.8 Moniteur de sécurité optimisé pour le suivi de flux d'information

Le cœur du moniteur de sécurité, présenté par la Figure 2.20, est chargé d'exécuter un *firmware*, de communiquer avec les différentes interfaces, de propager, de combiner et de vérifier les niveaux de sécurité traités.

FIGURE 2.20 – Cœur du moniteur

Ce cœur est basé sur un microprocesseur RISC MIPS 32 bits, cadencé à 150 MHz, qui a été modifié à partir du projet Plasma MIPS [6]. Le cœur du moniteur possède plusieurs interfaces permettant de communiquer avec la partie système via le bus AXI :

FIRMWARE_CODE_DATA est connectée à la mémoire Block RAM Memory (BRAM) stockant les instructions et les données du *firmware* s'exécutant sur le moniteur ;

PTM permet de lire les traces provenant du PTM via une FIFO.

MONITOR_TO_KERNEL est le canal de communication du moniteur vers le système d'exploitation. C'est via ce canal que le moniteur envoie par l'intermédiaire d'une FIFO, les étiquettes demandées par le module de sécurité utilisant les crochets LSM.

KERNEL_TO_MONITOR est le canal de communication permettant au système d'exploitation d'envoyer des requêtes ou des informations au moniteur de sécurité via une FIFO.

INSTRUMENTATION est connecté à la FIFO d'instrumentation contenant les différentes valeurs envoyées par l'application grâce à l'instrumentation du code. Lorsque le moniteur a besoin d'une donnée provenant de l'application, il vient lire les données depuis ce canal.

BBT_ANNOT est connecté à la mémoire BRAM contenant la table des blocs de base (BBT) ainsi que les annotations du programme.

6. https://opencores.org/projects/plasma

SEC_POLICY est l'interface permettant de configurer la politique de sécurité du moniteur à l'aide d'une mémoire BRAM.

INTERRUPT est la ligne d'interruption permettant au moniteur de stopper l'exécution de l'application et d'alerter le système d'exploitation qu'un problème est survenue lors de la propagation des étiquettes.

PL_TO_PS est l'interface connectée au gestionnaire d'interruption permettant de fournir des informations sur les raisons de l'interruption qui a été provoquée à la partie système (PS).

Le stockage des étiquettes par le moniteur est séparé en plusieurs mémoires selon les types d'étiquettes à stocker :

— les étiquettes correspondantes aux registres CPU sont stockées dans le TRF ;
— les étiquettes relatives aux zones mémoire sont-elles stockées dans le TM.

La Figure 2.21 présente la machine à états finis implémentée dans le *firmware* du moniteur. Lorsqu'un utilisateur souhaite exécuter une application et y appliquer un contrôle de flux d'informations, la première étape est de charger, dans la mémoire réservée au moniteur, les annotations qui sont stockées dans des segments du fichier exécutable ELF. La deuxième étape est de configurer la politique de sécurité au niveau du moniteur. Pour cela, une interface de configuration est proposée par le moniteur. Le moniteur se met alors en attente d'une adresse décodée par le *PFT decoder* et provenant du PTM. Dès qu'une trace est reçue, le moniteur se met à la recherche des annotations correspondant au bloc de base de l'adresse reçue dans la trace PTM et les exécute une par une. Selon le type de l'annotation, une communication avec la PS sera alors nécessaire ou non. Deux cas requièrent une communication avec la PS :

— lors de l'exécution de l'annotation $waiting\_instrumentation(Rs)$, qui se met en attente d'une donnée depuis la FIFO d'instrumentation ;
— lorsqu'une annotation $waiting\_value\_from\_lsm()$ est exécutée. , cette annotation correspond à un appel système qui se produit côté système, il est donc nécessaire d'attendre une réponse de la part du système d'exploitation.

Lorsqu'une violation de politique de sécurité ou de dépassement de capacité des mémoires survient, une interruption est générée par le moniteur de sécurité pour en informer le système d'exploitation. Puisque les sources et les raisons de générer une interruption peuvent être nombreuses , un gestionnaire d'interruption appelé « PL_TO_PS » permet de lever une interruption sur le processeur. Ce composant possède également de la mémoire qui est utilisée par le moniteur pour y écrire les raisons de l'interruption, cette mémoire peut

être lue par le système d'exploitation, ce qui permet de réagir différemment selon les raisons de l'interruption.

2.9 Conclusion

La mise en œuvre de cette approche a demandé un nombre conséquent de modifications de toutes les couches logicielles du système, en plus de la création d'un moniteur de sécurité spécialisé pour le contrôle de flux d'information. Les différentes transformations du code, l'instrumentation et les analyses statiques ont été implémentées dans le back-end *ARM* du framework LLVM. L'éditeur de liens *GNU ld* a également fait les frais de modifications pour la création du Basic Block Table et la résolution des symboles utilisés par celui-ci. La bibliothèque standard C (musl [7]) a bénéficié du support du DIFT. L'activation de tous les composants nécessaires au bon fonctionnement du DIFT a été en partie réalisée dans cette bibliothèque standard, au même titre que les solutions standards de sécurité telles que les *stack canaries*. Le développement des protocoles de communication et de synchronisation entre le moniteur et le système d'exploitation a représenté un travail d'implémentation conséquent dans le noyau du système d'exploitation et dans le *firmware* du moniteur de sécurité. L'ensemble de ces modifications permet d'effectuer du suivi de flux d'information dans les différentes couches du système. Afin d'évaluer la capacité de détection de cette approche, l'élaboration de politiques de sécurité est nécessaire et fera l'objet du chapitre suivant.

7. https://www.musl-libc.org

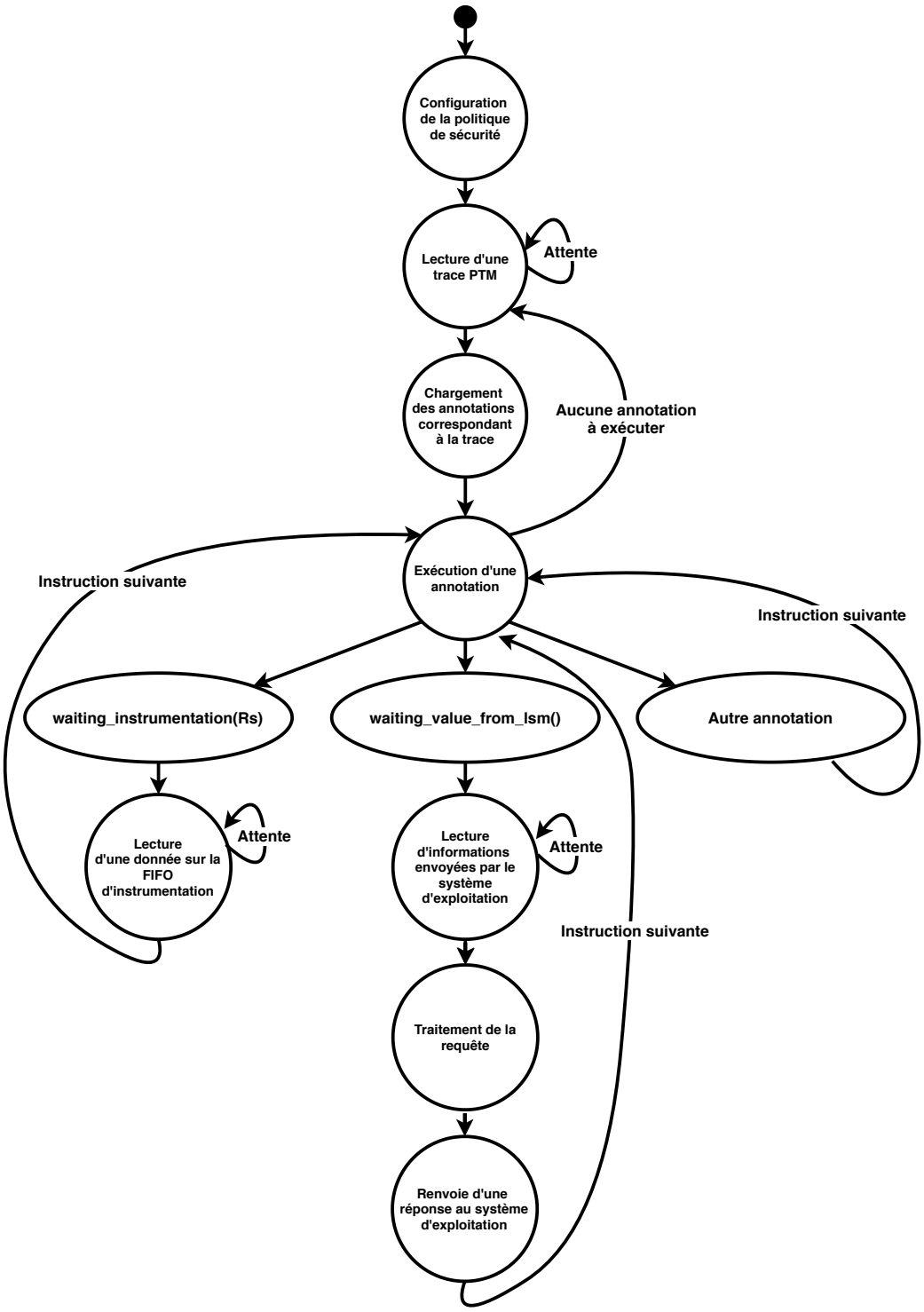


FIGURE 2.21 – Machine à état fini du moniteur

Chapitre 3

Politiques de sécurité et résultats expérimentaux

Notre moniteur de sécurité a été développé comme un système de détection d'intrusion permettant de garantir des propriétés de confidentialité et d'intégrité tout au long de l'exécution d'un programme. Ainsi, le moniteur de sécurité doit être capable de détecter l'exploitation de vulnérabilités connues ou non. Notre objectif est que l'administrateur du système soit alerté lors de la détection de violation d'une politique de sécurité, même tardivement, afin qu'il puisse agir en conséquence. La définition des politiques de sécurité est donc un élément déterminant pour détecter les compromissions de la sécurité du système. L'objet de ce chapitre est de présenter les politiques de sécurité basées sur les flux d'information qui permettent de détecter l'exploitation de vulnérabilités dans un système. Ce chapitre est organisé comme suit :

— la Section 3.1 détaille comment une politique de sécurité peut être exprimée avec notre moniteur de sécurité.
— la Section 3.2 montre comment un utilisateur peut protéger ses données, en matière de confidentialité en évitant toute fuite d'information, et d'intégrité en empêchant la corruption de ses fichiers.
— la Section 3.3 propose différentes politiques de sécurité qui peuvent être utilisées et les résultats obtenus en termes de détection.
— la Section 3.4 évalue l'impact sur les performances liées à chaque politique de sécurité.

3.1 Expression des politiques de sécurité

Une politique de sécurité spécifie de façon claire et précise les états du système qui sont autorisés et ceux qui sont interdits. Puisque notre moniteur

de sécurité est basé sur les flux d'information, il est alors nécessaire de définir des politiques de sécurité en termes de flux d'information. La définition d'une politique de flux d'information concerne trois étapes distinctes :

1. l'**initialisation** d'étiquettes consiste à initialiser le système en affectant une étiquette à certains conteneurs d'information afin de refléter le niveau de sécurité des informations contenues ou provenant de ces conteneurs.
2. la **propagation** de la valeur de l'étiquette d'un conteneur d'information source vers un conteneur d'information destination est réalisée lors de la manipulation des conteneurs d'information par les instructions machine de l'application, les règles de propagation ont été définies dans le Chapitre 2
3. la **vérification** permet de s'assurer que la valeur d'une étiquette correspond à la valeur attendue. En cas de violation de la politique, le système lève une interruption pour alerter l'administrateur système.

Notre souhait est de pouvoir détecter un large spectre d'attaques. En effet notre approche permet de suivre à la fois les flux internes à une application ainsi que les flux entre applications, via les conteneurs d'information gérés par l'OS. Cela permet de détecter les attaques à différentes étapes de leur réalisation. Par exemple, supposons qu'un attaquant exploite une vulnérabilité de type *buffer overflow* dans une application serveur afin de modifier son flot de contrôle et *in fine* faire fuir une base de mot de passe, contenue dans un fichier confidentiel, via le *socket* d'écoute du programme. Cette attaque peut être détectée à deux niveaux, car elle engendre deux types de flux d'information illégaux :

— En exploitant le *buffer overflow*, l'attaquant modifie l'adresse de retour d'une fonction stockée sur la pile (qui est une donnée ayant un fort besoin d'intégrité) en utilisant des données issues du *socket* d'écoute du serveur (et qui sont par nature d'un faible niveau de confiance) ;
— La fuite de données du fichier de mot de passe (possédant un fort niveau de confidentialité) vers le *socket* constitue un deuxième flux illégal.

Le premier type de flux est interne à l'application. La politique à définir est relativement générique : pour tous les programmes, il est nécessaire d'interdire les flux visant à modifier illégalement l'adresse de retour des fonctions. Le deuxième flux met en jeux des conteneurs d'information externes à l'application, et qui peuvent potentiellement être modifiés par d'autres applications. La politique à mettre en œuvre dépend du contexte de déploiement. Elle nécessite qu'un administrateur spécifie que le fichier de mot de passe est confidentiel.

Ces deux politiques sont complémentaires. En effet, l'attaquant pourrait également faire fuir le contenu du fichier de mot de passe en utilisant un autre type de vulnérabilité, potentiellement inconnu, qui ne serait pas défini par le premier type de politique. À l'inverse, l'attaquant peut également détourner le flot de contrôle pour réaliser un autre type de comportement malveillant (par exemple un déni de service), qui ne serait pas détecté par la deuxième politique.

Puisque nous souhaitons détecter des problèmes de sécurité très différents, il est nécessaire de définir différentes politiques de sécurité. En particulier nous distinguons les politiques de sécurité garantissant la confidentialité et l'intégrité de l'exécution d'une application, qui fera l'objet de la Section 3.3, de la politique de sécurité définie par l'administrateur du système permettant de contrôler les flux d'information entre les différents conteneurs d'information de l'OS, qui sera détaillée dans la Section 3.2.

Pour conséquent, la représentation des étiquettes a dû être divisée en deux comme illustré dans la Figure 3.1. La première partie, nommée *SYSTEM_POLICY* est réservée à l'intégrité et à la confidentialité de l'application, et la deuxième partie, appelée *USER_POLICY*, représente les niveaux de sécurité spécifiés par l'administrateur du système pour garantir la sécurité du système. Notre moniteur DIFT permet d'appliquer ces deux types de politiques de sécurité. Plus précisément, une étiquette est un mot de 32 bits découpé comme suit :

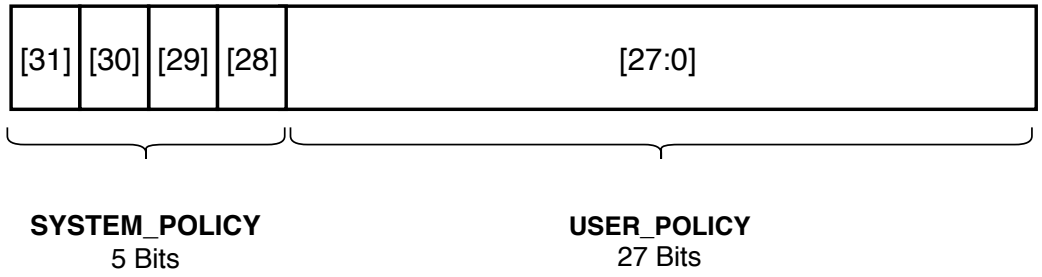


FIGURE 3.1 – Représentation d'une étiquette

Le bit 31 nommé *RET_ADDR_CFI*, est utilisé pour l'intégrité du flot de contrôle et sera traité plus en détail dans la Section 3.3.1.

Le bit 30 nommé *ALLOC_HEAP_MEMORY*, est réservé à l'étiquetage des zones mémoire allouées sur le tas lors d'un appel à la fonction *malloc* de la bibliothèque standard C et fera l'objet de la Section 3.3.2.

Le bit 29 nommé *RELEASED_HEAP_MEMORY*, est réservé à l'étiquetage des zones mémoire sur le tas qui ont été libérées via un appel

à la fonction *free* de la bibliothèque standard C et fera l'objet de la Section 3.3.2.

Le bit 28 nommé *SUSPICIOUS_INPUT*, permet d'identifier les informations provenant de sources de défiance. On utilisera ce bit lorsque l'application veut vérifier qu'une donnée utilisée pour une opération sensible n'a pas été introduite par un attaquant. La Section 3.3.3 décrit la politique de sécurité ainsi que l'utilisation de ce bit.

Les bits 27 à 0 sont utilisés librement par l'utilisateur pour étiqueter des conteneurs d'information. La Section 3.2 illustrera comment ces bits peuvent être utilisés pour garantir des propriétés de sécurité dans tout le système.

Afin d'identifier les différentes parties d'une étiquette, on utilise les notations suivantes :

— $\underset{USER}{X}$ fait référence à la partie *USER_POLICY* d'une étiquette ;
— $\underset{SYSTEM}{X}[I]$ fait référence au bit I de la partie *SYSTEM_POLICY*.

Pour pouvoir modifier les étiquettes des différents conteneurs d'information du système, l'administrateur système dispose de plusieurs outils permettant de configurer la politique de sécurité souhaitée. Il est ainsi nécessaire de définir, pour chaque politique de sécurité, les valeurs par défaut des étiquettes qui seront utilisées lors de la phase d'initialisation, les opérations utilisées pour la propagation des étiquettes et les vérifications à réaliser pour s'assurer que la politique de sécurité est respectée.

L'administrateur système peut ainsi préciser les opérations binaires utilisées pour chaque type d'instruction présentée dans l'équation 2.7. Il est possible d'utiliser les opérations bit à bit usuelles d'arithmétique et de logique suivantes pour la propagation des étiquettes :

— *OR* : (la fonction OU logique)
— *NOR* : (la fonction NON-OU logique)
— *AND* : (la fonction ET logique)
— *NAND* : (la fonction NON-ET logique)
— *XOR* : (la fonction ou exclusif)
— *XNOR* : (l'équivalence logique)
— *SUM* : (l'addition arithmétique)
— *SUB* : (la soustraction arithmétique)
— *SUP* : (l'infériorité arithmétique)
— *INF* : (la supériorité arithmétique)

Un fichier nommé *hardblare_policy.conf*, présent dans le répertoire *etc* de la racine du système de fichiers, permet de fixer les opérations à utiliser lors de la propagation des étiquettes pour chaque politique de sécurité. L'administra-

teur est responsable de cette configuration, le choix des opérations utilisées doit assurer l'associativité et la commutativité. Nous ne traiterons pas ici comment fixer les opérations pour obtenir des représentations sous forme de structures mathématiques par exemple des treillis. En effet, ce point est hors champ de cette thèse. La grammaire du fichier de configuration est définie par le code 3.1.

```
type ::= ARITH_LOGIC
       | MOVE
       | COMPARE
       | ARITH_LOGIC_AND_COMPARE
       | BRANCH

op ::= OR
       | NOR
       | AND
       | NAND
       | XOR
       | XNOR
       | SUM
       | SUB
       | SUP
       | INF

rule ::= type op

config ::= rule
       | config
```

CODE SOURCE 3.1 – Grammaire du fichier hardblare_policy.conf

Le code 3.2 illustre un exemple du contenu possible pour ce fichier. L'administrateur a alors choisi d'utiliser : le « ou logique » pour propager les étiquettes issues des instructions arithmétiques et logiques avec et sans mise à jour des champs du registre APSR, le « et logique » pour les annotations issues des instructions de mouvement des données, « l'addition » pour les annotations générées pour les instructions de comparaisons, et la supériorité arithmétique pour les annotations issues des instructions de branchements.

```
0  $ cat /etc/hardblare_policy
1      ARITH_LOGIC OR
2      MOVE AND
3      COMPARE SUM
```

```
4    ARITH_LOGIC_AND_COMPARE OR
5    BRANCH SUP
```

CODE SOURCE 3.2 – Contenu du fichier hardblare_policy.conf

Comme décrit en section 2.6.1, RfBlare permet d'associer deux étiquettes à chaque fichier :

- *iTag* est l'étiquette qui reflète le niveau de sécurité du contenu courant du fichier
- *pTag* est l'ensemble d'étiquettes permettant de spécifier le niveau de sécurité autorisé par la politique de flux, pour le contenu de ce fichier.

L'étiquette *iTag* est amenée à évoluer lorsque des exécutables modifient le contenu du fichier, afin de refléter le niveau de sécurité du nouveau contenu. Toutefois, lors de la mise en place du DIFT, l'administrateur doit étiqueter une première fois chacun des conteneurs [1] pour spécifier le niveau de sécurité des information qu'il contient.

L'ensemble d'étiquettes *pTag* n'est pas propagée. Elle est représenté par un ensemble de niveaux de sécurité. Elle n'évolue pas sauf si l'administrateur décide de modifier la politique de sécurité. Elle est en revanche utilisé lors de la vérification de la politique de sécurité. Un flux d'information à destination d'un fichier est légal si l'étiquette *iTag* résultant de ce flux est dans l'ensemble *pTag* d'étiquettes ($iTag \in pTag$).

Pour pouvoir agir sur les étiquettes *iTag* et *pTag*, trois outils hérités du projet RfBlare [1] ont été adaptés pour HardBlare. Les deux premiers outils permettent de modifier l'étiquette *iTag* d'un fichier :

- `getinfo` permet d'accéder à l'intégralité de l'étiquette *iTag* stockée dans l'attribut étendu du fichier ;
- `setinfo` permet d'affecter ou de modifie directement l'intégralité de l'étiquette *iTag* stockée dans les attributs étendus du fichier.

Le code 3.9 illustre l'utilisation de ces deux commandes Linux.

```
0    $ ./getinfo notes.txt
1        1 iTag detected
2        iTag = 0x00001352
3
4    $ ./setinfo input.txt 0x41
5        iTag value = 0x41
```

1. Initialement il n'est pas nécessaire d'étiqueter tous les conteneurs mais seulement ceux dont on souhaite suivre le contenu. L'absence d'étiquette correspond à un contenu « public ».

```
6    Successfuly wrote iTag
```

CODE SOURCE 3.3 – Exécution normale de l'application

Le troisième outil, `setpolicy` permet quant à lui de modifier le *pTag* d'un fichier. Le code 3.9 illustre l'utilisation de cette commande.

```
0    $ ./setpolicy notes.txt {0x00001234; 0x00001352}
1        pTag = {0x00001234; 0x00001352}
2        Successfuly wrote pTag
```

CODE SOURCE 3.4 – Exécution normale de l'application

On notera que les commandes `setpolicy` et `setinfo`, qui permettent de modifier la politique de flux, ne doivent être accessibles qu'à un administrateur. Dans notre implémentation, elles ne sont accessibles qu'à l'utilisateur *root*. Il s'agit d'une protection minimale qui pourrait être complétée par d'autres restrictions. Par exemple, on pourrait envisager de rendre ses commandes accessibles uniquement dans un mode spécifique d'administration accessible uniquement au démarrage de l'ordinateur.

Les moyens de configuration d'une politique de sécurité présentées ci-dessus permettent de spécifier concrètement les politiques de sécurité présentées dans les Section 3.3 et 3.2.

Dans la section suivante, nous détaillons comment l'administrateur système peut contrôler la propagation des données dans le système, en configurant le moniteur de sécurité et en affectant des étiquettes aux fichiers du système.

3.2 Protection des fichiers du système à l'aide d'une politique de sécurité

La structure des étiquettes présentée par la Figure 3.1 offre 28 bits à l'administrateur système pour représenter les niveaux de sécurité d'une politique de flux d'information dans le système. La politique de sécurité consiste donc à attacher des étiquettes *iTag* et *pTag* aux fichiers du système et à configurer les opérations utilisées lors de la propagation des étiquettes, dans le fichier *hardblare_policy.conf*. Prenons l'exemple de l'application 3.5, qui lit des données du fichier *input.txt* à l'aide de la variable *buffer_input_file* et les données du fichier *notes.txt* à l'aide de la variable *buffer_notes_file*.

De façon conditionnelle, soit les données du fichier *buffer_input_file*, soit les données du fichier *buffer_notes_file* sont ensuite transférées dans la variable *buffer*. Les données présentes dans la variable *buffer* sont ensuite écrites dans le fichier *fuite.txt*.

```
0  #include <stdio.h>
1  #include <fcntl.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6
7    int fd_input, fd_fuite, fd_notes, ret_input, ret_notes, i;
8    char buffer_input_file[50];
9    char buffer_notes_file[50];
10   char buffer[50];
11
12   fd_input = open("input.txt", O_RDONLY);
13   fd_notes = open("notes.txt", O_RDONLY);
14   fd_fuite = open("fuite.txt", O_WRONLY | O_APPEND);
15
16   if (fd_notes < 0) {
17     perror("Error notes.txt \n"); exit(1);
18   }
19   if (fd_input < 0) {
20     perror("Error input.txt \n"); exit(1);
21   }
22   if (fd_fuite < 0) {
23     perror("Error fuite.txt \n"); exit(1);
24   }
25
26   ret_input = read(fd_input, buffer_input_file, sizeof(buffer_input_file));
27
28   if (ret_input < 0) {
29     perror("Error when reading \n"); exit(1);
30   }
31
32   ret_notes = read(fd_notes, buffer_notes_file, sizeof(buffer_notes_file));
33
34   if (ret_notes < 0) {
35     perror("Error when reading \n"); exit(1);
36   }
37
38   if(argc == 1){
39         for(i = 0; i < sizeof(buffer_notes_file); i++) {
40               buffer[i] = buffer_notes_file[i];
41         }
42         write(fd_fuite, buffer, ret_notes);
43   }
```

```
44    else{
45          for(i = 0; i < sizeof(buffer_input_file); i++) {
46                buffer[i] = buffer_input_file[i];
47          }
48          write(fd_fuite, buffer, ret_input);
49    }
50
51    close(fd_notes);
52    close(fd_input);
53    close(fd_fuite);
54  }
```

CODE SOURCE 3.5 – Exemple d'application pour laquelle on désire une propriété de non-interférence

La première étape de l'administrateur système est d'affecter un niveau de sécurité au fichier *input.txt*, ici la valeur *0x5*. L'administrateur affecte ensuite le niveau de sécurité *0x42* au fichier *notes.txt*. Ainsi lors de la manipulation des données de ces fichiers, leurs niveaux de sécurité seront propagés. L'administrateur système souhaite que seules des données étiquetées avec le niveau de sécurité *0x5* soient présentes dans le fichier *fuite.txt*. Aucune donnée provenant du fichier *notes.txt* ne doit donc être propagée vers le fichier *fuite.txt*.

```
0   $ ./setinfo input.txt 0x5
1       Tag value = 0x5
2       Successfuly wrote tags
3
4   $ ./setinfo notes.txt 0x42
5       Tag value = 0x42
6       Successfuly wrote tags
7
8   $ ./setpolicy fuite.txt {0x5}
9       pTag = {0x5}
10      Successfuly wrote pTag
```

CODE SOURCE 3.6 – Exécution normale de l'application

Ainsi, lors de l'exécution de l'application, si l'application essaye d'écrire dans le fichier *fuite.txt* avec des données provenant du fichier *notes.txt*, le crochet LSM détecte une violation de politique de sécurité car le tag *0x42* ne

fait pas partie de l'ensemble de $pTag$ du fichier *fuite.txt*. Le code 3.7 présente le résultat de l'exécution de l'application dans le cas cité précédemment.

```
0  $ ./read-write.elf
1      [HardBlare LSM Hook (hardblare_file_permission)]:
2          File = /home/user/input.txt
3          Action = READ
4          Sending file's tag and memory area to the Monitor :
5              File.tag: 0x5
6              *buff: 0xbee83c18
7              size: 0x32
8
9      [HardBlare LSM Hook (hardblare_file_permission)]:
10         File = /home/user/notes.txt
11         Action = READ
12         Sending file's tag and memory area to the Monitor :
13             File.tag: 0x42
14             *buff: 0xbee83c4a
15             size: 0x32
16
17     [HardBlare LSM Hook (hardblare_file_permission)]:
18         File = /home/user/fuite.txt
19         Action = WRITE
20         Requesting memory area's tag to the Monitor :
21             *buff: 0xbee83c7c
22             size: 0x32
23         Tag recevied from monitor:
24             [0xbee83c7c; ...; 0xbee83c7c + 0x32] = 0x42
25         [ERROR] when trying to affect isec->info.tags[
               ↪ BLARE_TAGS_NUMBER_IDX] = 0x42
26
27     [HardBlare] Policy violation for /home/user/fuite.txt: iTag 0x42 is
           ↪ not in pTag {0x5}
```

CODE SOURCE 3.7 – Exécution normale de l'application

La Figure 3.2 illustre les flux d'information que se produisent lors de l'exécution du programme. On peut alors remarquer que la propagation des étiquettes se fait de façon fine dans la mémoire de l'application, là où RfBlare[1] aurait étiqueté toute la mémoire de l'application avec une combinaison des étiquettes des deux fichiers *input.txt* et *notes.txt*.

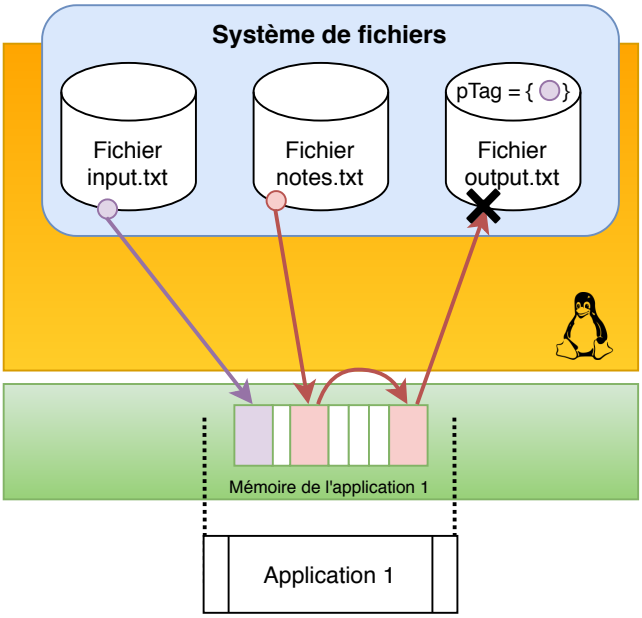


FIGURE 3.2 – Propagation des étiquettes dans le système et détection d'une violation de politique

HardBlare propose donc une solution permettant d'effectuer un contrôle de flux d'information entre les conteneurs d'information de différente granularité. Nous laissons ainsi la liberté à l'administrateur système de développer ses propres politiques de sécurité et la représentation des niveaux de sécurité qu'il souhaite. Dans la section suivante, nous détaillons les politiques de sécurité développées permettant d'assurer la sécurité de l'application.

3.3 Protection de l'exécution de l'application

Le fonctionnement d'une application peut être corrompu de plusieurs façons. Dans cette thèse, nous avons évalué notre approche sur trois types de problèmes de sécurité affectant la sécurité de l'application.

L'atteinte à l'intégrité du flot de contrôle se produit par exemple lorsqu'un attaquant réussit à modifier l'adresse de retour d'une fonction lui permettant ainsi de prendre le contrôle des prochaines instructions exécutées.

Les implémentations modernes des OS ne permettent plus à un attaquant de modifier en mémoire le code des applications ni d'exécuter directement des données. L'attaquant peut toutefois modifier le flot d'exécution pour faire exécuter à l'application un enchaînement de parties du code existant, appelées *gadgets*. Cette attaque est appelée Return Oriented Programming. Pour pouvoir se protéger de ce type d'attaque, il est nécessaire de mettre en place une solution d'intégrité du flot de contrôle, appelé Control Flow Integrity (CFI) en anglais. Cette politique de sécurité fait l'objet de la Section 3.3.1.

La corruption de la mémoire peut se produire lorsqu'un attaquant tente d'accéder à une zone mémoire après qu'elle ait été libérée via un appel à la fonction « free », cette vulnérabilité porte le nom de Use After Free. Elle peut également se produire lors d'un double appel à la fonction « free ». La Section 3.3.2 traite de ces deux vulnérabilités.

L'utilisation de données provenant d'une source contrôlée par l'attaquant en entrée d'une opération sensible de l'application. Cette corruption est souvent la cible des travaux sur la propagation des teintes [25]. C'est par exemple le cas lors d'une injection de commandes SQL ou lorsqu'un utilisateur peut contrôler le formatage de chaînes de caractères utilisé par la fonction `printf`. La Section 3.3.3 illustre l'exemple d'une application détectant les injections SQL.

Chaque cas cité précédemment fait l'objet d'une politique de sécurité spécifique permettant de détecter ces attaques. L'administrateur système peut selon sa volonté, activer ou désactiver chaque politique de sécurité de façon indépendante lors de la compilation de l'exécutable. Ses politiques sont décrites dans les sous- sections suivantes. L'évaluation des performances sera détaillée dans la Section 3.4.

3.3.1 Politique de sécurité pour l'intégrité du flot de contrôle

L'utilisation d'un même espace mémoire pour stocker à la fois des données de contrôle nécessaires pour le bon fonctionnement de l'application et des données utilisateurs, sans isolation stricte, peut permettre à un attaquant d'agir sur des données de contrôle et donc modifier le comportement de l'application. Assurer l'intégrité du flot de contrôle d'une application est un problème complexe qui nécessite de vérifier les adresses cibles des sauts effectués, mais également les adresses utilisées par les sous-routines afin de retourner vers le code appelant.

Pour que le moniteur puisse décider si une violation de l'intégrité du flot de contrôle s'est produite, le moniteur doit connaitre les adresses de saut

valide.

Dans le graphe de flot de contrôle d'une application, on parlera « d'arcs avant » lorsqu'un saut ou un branchement est exécuté, et « d'arcs arrières » lorsqu'il s'agit d'un retour vers le code appelant. Par exemple, la Figure 3.3 illustre le chemin pris lors de l'appel de *f(g(h(i())))* et des retours successifs.

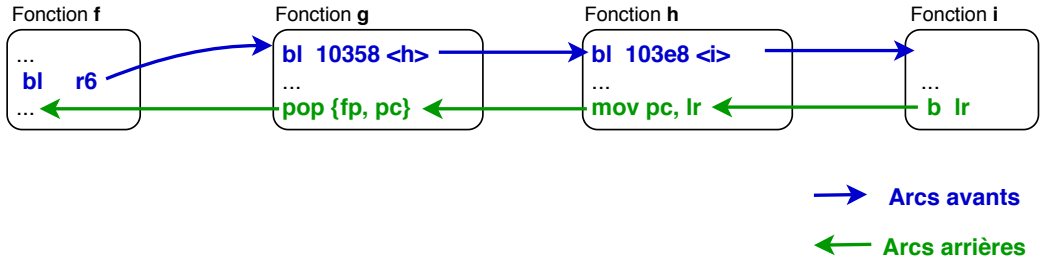


FIGURE 3.3 – Arcs avants et arrières dans le flot de contrôle

Il existe deux types d'instructions dans l'architecture ARMv7 pour contrôler le flot d'exécution. Les instructions « d'appel » qui permettent à une fonction appelante d'appeler une autre fonction, ce sont par exemple les instructions *b* et *bl* ainsi que leurs variantes conditionnelles, les instructions ayant le registre PC comme registre destination (*add pc, ...*) ; et des instructions « de retour » qui permettent à une fonction appelée de retourner dans la fonction appelante, par exemple *mov pc, lr* ou *pop {.., pc}*.

Lors de l'exécution d'une application, plusieurs fonctions peuvent ainsi être appelées en cascade. Lors d'un appel, l'ABI spécifie que la fonction appelante sauvegarde, dans le registre LR, l'adresse à laquelle la fonction appelée doit retourner lorsqu'elle a terminé son exécution. Comme il n'y a qu'un seul registre, ce registre est empilé sur la pile puis dépilé pour pouvoir permettre des appels en profondeur. Le problème principal de cette méthode est que des adresses de retour utilisées pour contrôler le flot de contrôle de l'application se retrouvent sur la pile de données de l'application qui peuvent être modifiées. La question à se poser est donc : *comment s'assurer qu'un attaquant ne puisse pas modifier une adresse de retour sur la pile ?*

Une des solutions est de sceller les adresses de retour stockées sur la pile afin d'éviter toute modification. L'administrateur doit alors spécifier lors de la compilation qu'il souhaite générer les annotations, les structures de données et mener les vérifications nécessaires dans le moniteur de sécurité pour pouvoir garantir l'intégrité du flot de contrôle.

Lors d'un branchement avec l'instruction *bl* pour « Branch and Link », l'adresse qui succède l'instruction de branchement est sauvegardée dans le registre LR puis la valeur du registre PC est remplacé par l'adresse cible de

la fonction appelée.

Il est donc nécessaire de créer une annotation supplémentaire lors de la compilation à chaque rencontre de l'instruction *bl*, afin d'initialiser (mettre à 1) le bit RET_ADRR_CFI dans la partie $SYSTEM\_POLICY$ de l'étiquette du registre LR

`bl printf`	$\underset{SYSTEM}{lr}[RET\_ADRR\_CFI] = 1$

La propagation de l'étiquette entière du registre LR se produit lors de sa sauvegarde sur la pile. Aucune annotation supplémentaire n'est nécessaire lors de sa propagation.

Par contre, toute modification ou mélange de l'étiquette de l'adresse de retour sur la pile remettra à 0 le registre RET_ADRR_CFI de l'étiquette car seule l'exécution d'un branchement permet de mettre d'initialiser la valeur RET_ADRR_CFI dans une étiquette. Seules les instructions de mouvement de données, d'écriture et de lecture sur le registre entier de 32 bits préservent le bit RET_ADRR_CFI.

Ainsi, lors de la compilation, l'analyse statique va détecter toutes les instructions utilisant le registre LR ou une adresse sur la pile pour mettre à jour le registre PC (instructions de « retour »), et va générer une annotation afin de vérifier l'intégrité du bit RET_ADRR_CFI. Ainsi, si l'étiquette du registre LR ou de l'adresse utilisée pour mettre à jour le registre PC n'est plus scellée avec le bit RET_ADRR_CFI, une interruption va être générée pour prévenir l'administrateur.

Les deux instructions suivantes illustrent à droite la vérification effectuée en pseudo-code par le moniteur de sécurité.

`mov pc, lr`	$if(\underset{SYSTEM}{lr}[RET\_ADRR\_CFI] == 0)\{$ $raise\_interrupt\_policy\_violation()$ $\}$

pop {..., pc}	$if(\underbrace{address(value(sp)+4)}_{SYSTEM}\ [RET\_ADRR\_CFI] == 0)\{$ $raise\_interrupt\_policy\_violation()$ }

Dans le cas des arcs avant, il est nécessaire de vérifier que l'adresse cible corresponde à une adresse prévue. Les appels peuvent être des branchements ou des sauts soit directs, soit indirects.

Lorsque les appels se font de façon directe, l'adresse de branchement est facile à déterminer statiquement. De plus, après les transformations du code de l'application détaillée dans la Section 2.3, chaque bloque de base utilisant un branchement direct peut avoir au maximum deux arcs avants possibles. Ces adresses sont directement présentes dans le code de l'application qui est normalement protégé en écriture lors de son exécution. Cependant, dans le cas où le code de l'application peut être modifié [2], notre approche peut permettre de protéger les adresses cibles. Ces adresses sont alors stockées dans le fichier exécutable dans un nouveau segment nommé *HARDBLARE_CFI* qui contient la section *.HB.cfi* comme illustré dans la Figure 3.5. Cette section stocke, pour chaque adresse de bloc de base, le nombre d'adresses cibles possibles ainsi que leurs valeurs. Par exemple, le bloc de base ayant l'adresse *0x10540* possède un seul arc avant qui a pour cible l'adresse *0x10470*. Le bloc de base *0x10520* contenant un branchement indirect, possède deux arcs avants possibles ciblant soit l'adresse *0x10328*(*english()*) soit l'adresse *0x10358*(la fonction *french()*). Ces informations sont référencées par le Basic Block Table.

Ainsi lors de la compilation d'une application, l'utilisateur peut décider d'activer la génération de cette section *ELF*. La vérification est alors effectuée par le moniteur de sécurité, qui pour chaque trace PTM reçue, vérifie que la trace PTM suivante est bien dans l'ensemble des cibles potentielles indiquées par la section *.HB.cfi*. la figure 3.4 présente comment la vérification est menée par le moniteur de sécurité.

Lorsque l'appel se fait de façon indirecte, par exemple lors de l'utilisation d'un pointeur de fonction, un branchement indirect est réalisé comme dans la fonction f de la Figure 3.3. Lorsque les blocs de base possèdent des branchements indirects, le développeur doit indiquer les adresses cibles potentielles à l'aide de la macro *INDIRECT_BRANCH_CFI* qui permet de créer une

2. En cas d'erreur de la mise en place de cette protection par le noyau, ou si l'application rend les sections de code modifiables par l'intermédiaire de la fonction POSIX *mprotect*, par erreur ou suite à une précédente attaque

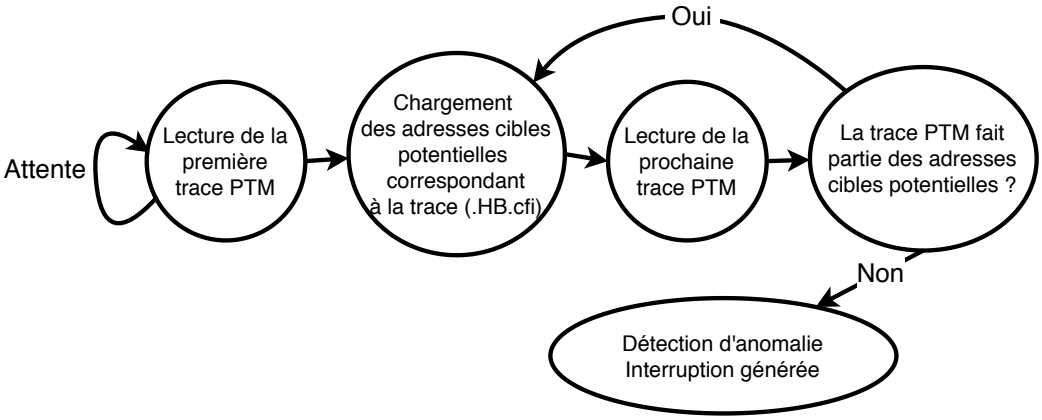


FIGURE 3.4 – Machine à états finis permettant l'intégrité du flot de contrôle

entrée dans la section *.HB.cfi* du fichier ELF en utilisant le symbole des fonctions passées en argument qui seront par la suite résolus par l'éditeur de lien. Dans l'exemple de la Figure 3.3, cela permet de connaître les adresses qui peuvent potentiellement être stockées dans le registre *r6*. La ligne 33 du code 3.8 donne un exemple de l'utilisation de cette macro.

Différentes approches ont été proposées dans la littérature afin de découvrir automatiquement les adresses de destination potentielles des branchements indirects. Par exemple, une analyse de pointeurs peut être utilisée. Toutefois, selon la nature du code de l'application, ce type d'analyse peut réaliser une surapproximation importante de l'ensemble des adresses destination de chaque branchement. Une première approche peut consister à autoriser les sauts vers tous les débuts de fonction du code de l'application, ce qui empêche le saut vers des *gadgets* situés à l'intérieur du code des fonctions. On peut aussi se restreindre aux fonctions ayant un prototype attendu. Ce type d'analyse peut être mis en œuvre lors de la compilation. Dans le cadre de cette thèse, nous avons focalisé nos travaux sur la vérification dynamique de cette politique et nous supposons que les macros *INDIRECT_BRANCH_CFI* sont correctement positionnées dans le code. Ce point de vue est partagé par d'autres travaux qui proposent des mécanismes permettant de vérifier une politique de type CFI à partir d'un support matériel [79].

Afin de vérifier que cette politique de sécurité détecte l'exploitation d'une vulnérabilité permettant à un attaquant de modifier une adresse de retour, nous avons testé notre approche sur l'exemple illustré par le Code 3.8. Cette application attend un nom sur l'entrée standard, et affiche un message de bienvenu dans une langue aléatoirement choisie entre l'anglais et le français.

L'utilisation de la fonction *scanf* à la ligne 32, permet à un attaquant

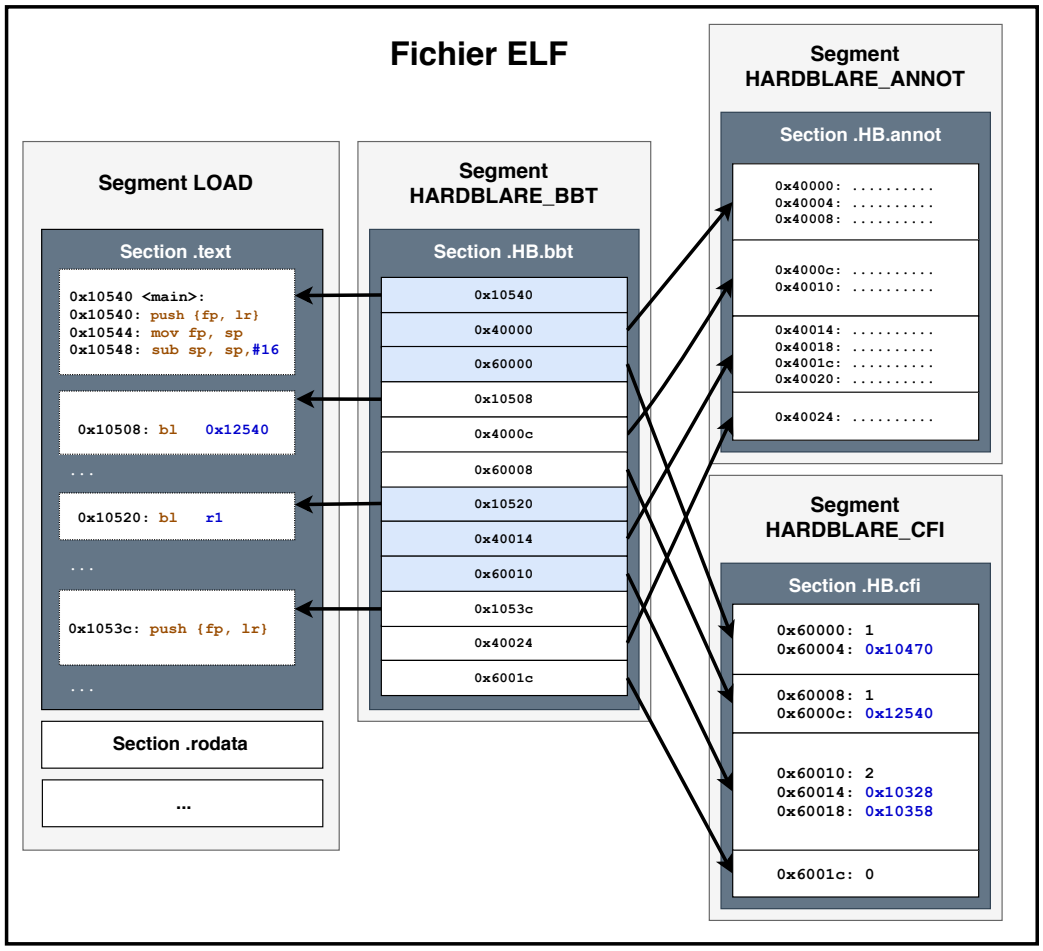


FIGURE 3.5 – Stockage des adresses cibles pour l'intégrité du flot de contrôle

d'écrire des données sur la pile à partir de l'adresse du tableau *buffer*. Ce tableau à une capacité de seulement quatre caractères, soit quatre octets. Cependant la fonction *scanf* ne limite en aucun cas le nombre d'octets que l'utilisateur peut fournir. La vulnérabilité présente est donc la possibilité d'écraser des données sur la pile en écrivant plus de quatre octets sur l'entrée standard.

Le but de l'attaquant est d'utiliser cette vulnérabilité afin de modifier les valeurs du tableau de pointeurs de fonction se trouvant à la ligne 26 pour dérouter l'appel de fonction de la ligne 33 vers la fonction *targetFunctionForward*. L'attaquant peut également modifier l'adresse de retour qui sera utilisée par la fonction *vuln* à la ligne 36 afin d'accéder à la fonction *targetFunctionBackward*.

Cette application permet donc de tester notre politique de sécurité concernant l'intégrité du flot de contrôle pour les « arcs avants » et les « arcs arrières ».

```
0  #include <stdlib.h>
1  #include <stdio.h>
2  #include <hardblare.h>
3
4  void english()
5  {
6    printf("Hello ");
7  }
8
9  void french()
10 {
11   printf("Bonjour ");
12 }
13
14 void targetFunctionForward()
15 {
16   printf("Hacked Forward!\n");
17 }
18
19 void targetFunctionBackward()
20 {
21   printf("Hacked Backward!\n");
22 }
23
24 void vuln()
25 {
26   void (*language[2])(void) = {english, french};
27   char buffer[4];
28   int randomLanguage;
29
30   srand(time(NULL));
31   randomLanguage = rand() % 5;
32   scanf("%s", buffer);
33   INDIRECT_BRANCH_CFI( language[randomLanguage]() , english, french);
34   printf(" %s!\n", buffer);
35
36   return;
37 }
38
39 int main(int argc, char **argv)
40 {
41   vuln();
42   return 0;
43 }
```

Code source 3.8 – Initialisation de l'instrumentation

Une fois l'application compilée et liée statiquement, celle-ci peut être exécutée de façon normale en entrant uniquement trois caractères sur l'entrée standard. Le Code 3.9 présente le résultat obtenu lors de son exécution avec la valeur « Foo » en entrée standard.

```
0  $ echo -n "Foo" | ./cfi.elf
1      Bonjour Foo!
```

CODE SOURCE 3.9 – Exécution normale de l'application

Le Code 3.10 présente l'exécution de l'application avec l'exploitation de la vulnérabilité. La première étape pour un attaquant est de découvrir l'adresse de la fonction ou du gadget qu'il souhaite utiliser. On récupère ici l'adresse des fonctions *targetFunctionForward* et *targetFunctionBackward* en lisant la table des symboles de l'exécutable. Une fois ces adresses connues, on stocke leurs valeurs dans un fichier nommé *payload.txt* [3]. L'exploitation des vulnérabilités consiste alors à exécuter l'application en lui donnant en entrée standard le contenu du fichier *payload.txt*, qui contient trois fois l'adresse de la fonction *targetFunctionForward*, puis deux fois l'adresse de la fonction *targetFunctionBackward*. On peut ainsi remarquer à la ligne 10 et 11 du Code 3.10, que les fonctions *targetFunctionForward* et *targetFunctionBackward* sont bien exécutées. Le moniteur de sécurité détecte l'exploitation de ces vulnérabilités en générant deux interruptions pour signaler à l'administrateur système que le bloc de base ayant pour adresse *0x00010520* a effectué un branchement vers une adresse imprévue, à savoir *0x000103e8*, et que le bloc de base ayant pour adresse *0x0001053c* a échoué lors de la vérification de l'étiquette à l'adresse *0x3FDC*, qui devait être scellée avec la valeur *RET_ADRR_CFI* car elle comportait une adresse de retour.

```
0  $ readelf -s cfi.elf
1    ...
2    000103e8 68 FUNC GLOBAL DEFAULT 2 targetFunctionForward
3    0001042c 68 FUNC GLOBAL DEFAULT 2 targetFunctionBackward
4
5  $ hexdump payload.txt
6    03e8 0001 03e8 0001 03e8 0001
7    042c 0001 042c 0001
8
```

3. On notera que les adresses sont inversées car l'architecture ARMv7 a une représentation Little-endian

```
$ cat payload | ./cfi.elf
  Hacked Forward!
  Hacked Backward!

  [HARDBLARE Driver] pl_to_ps_irq_handler:
    PLtoPS_Dispatcher_Status = 1
    PLtoPS_Dispatcher_DataSize = 2
    PLtoPS_Dispatcher_Data = 0x00010520
    Reason: Unexpected Branch Address (0x000103e8)

  [HARDBLARE Driver] pl_to_ps_irq_handler:
    PLtoPS_Dispatcher_Status = 1
    PLtoPS_Dispatcher_DataSize = 3
    PLtoPS_Dispatcher_Data = 0x0001053c
    PLtoPS_Dispatcher_Data = 0x3FDC
    Reason: 0x3FDC = Not sealled
```

CODE SOURCE 3.10 – Exploitation de la vulnérabilité de l'application

La Figure 3.6 présente en détail l'état de l'exécution de l'application lorsque la vulnérabilité est exploitée ou non. Dans cette figure, les valeurs scellées ont été coloriées en vert, les données non scellées provenant de l'entrée standard ont été coloriées en rouge. Les flèches bleues représentent les arcs avant et les flèches vertes représentent les arcs arrière lors de l'exécution normale de l'application. Les flèches rouges représentent quant à elles les violations d'intégrité du flot de contrôle lors de l'exploitation des vulnérabilités.

Le moniteur de sécurité vérifie donc que chaque trace PTM reçue correspond bien a une adresse cible potentielle conformément aux informations présentent dans la section *.HB.cfi* d'après la Figure 3.5. Le moniteur vérifie ensuite que l'adresse récupérée sur la pile à l'adresse *0x3FDC* est toujours scellée avant l'exécution de l'instruction *pop{fp, pc}* à l'adresse *0x1053c*.

Cette politique de sécurité permet donc de détecter les violations d'intégrité du flot de contrôle de l'application en temps réels, pour les « arcs avant » et pour les « arcs arrière » du graphe de flot de contrôle. Cette solution nécessite néanmoins une intervention de la part du développeur ou du compilateur de l'application afin d'indiquer les adresses cibles potentielles lors de l'utilisation de pointeurs de fonctions.

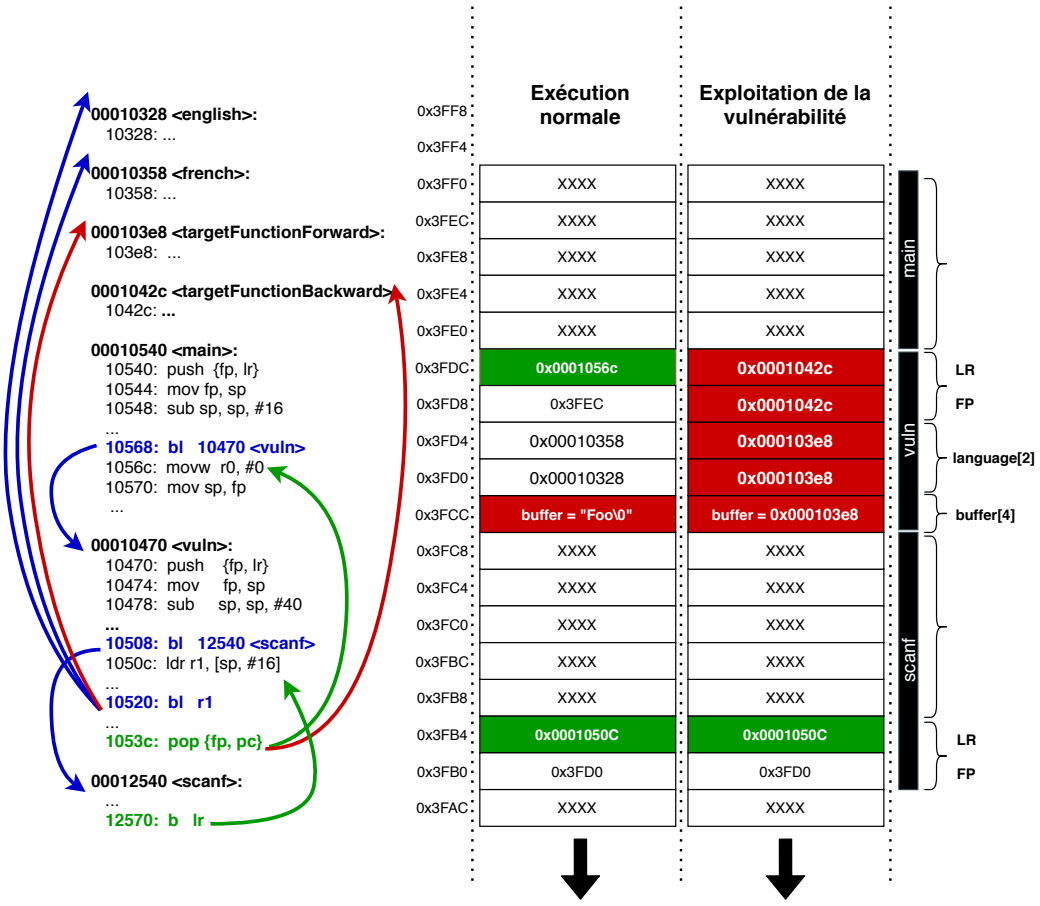


FIGURE 3.6 – Exemple de modification du flot de contrôle

3.3.2 Politique de sécurité pour les erreurs mémoires temporelles sur le tas

L'utilisation de la mémoire d'une application évolue de façon dynamique au cours de son exécution en allouant et en libérant de la mémoire sur la pile et sur le tas. Deux types d'erreurs peuvent se produire lorsqu'une adresse mémoire est manipulée :

Une erreur spatiale se produit lorsqu'une adresse pointe en dehors des bornes de l'objet mémoire qu'elle est censée référencer.

Une erreur temporelle se produit lorsque l'objet mémoire pointé par l'adresse a précédemment été libéré.

Nous proposons ici de s'intéresser aux erreurs temporelles sur le tas, qui sont difficiles à détecter à cause de leur aspect dynamique.

L'application présentée par le Code 3.11 présente deux erreurs temporelles sur le tas. La première erreur est que le programme essaye de déréférencer une adresse qui pointe vers une zone mémoire qui a été précédemment libérée. Cette vulnérabilité est connue sous la référence « CWE-416 - Use after free ». La deuxième erreur correspond à la double libération d'une zone mémoire, portant la référence « CWE-415 - Double free ». En 2020, ces deux vulnérabilités sont toujours très présentes parmi les failles découvertes et sont respectivement classées huitièmes et trente-huitièmes dans le classement des vulnérabilités les plus dangereuses par le MITRE [4].

```
0  #include <stdio.h>
1  #include <unistd.h>
2  #define BUFSIZE 262144
3
4  void useAfterFree(char *buf){
5    *buf = "Hello";
6  }
7
8  void doubleFree(char *buf){
9    free(buf);
10 }
11
12 int main(int argc, char **argv){
13   char *buf;
14   buf = (char *) malloc(BUFSIZE);
15
16   free(buf);
17
18   if(argc == 1) {
19     useAfterFree(buf);
20   }
21   if(argc == 2) {
22     doubleFree(buf);
23   }
24 }
```

CODE SOURCE 3.11 – Programme possédant les vulnérabilités CWE-415 et CWE-416

Pour pouvoir détecter l'exploitation de ces deux vulnérabilités, on doit définir une politique de flux de contrôle. Ainsi, la phase d'initialisation des étiquettes se produit lors de l'allocation d'une zone mémoire dans le tas par l'appel de la fonction *malloc*, on marque toute la zone mémoire en mettant à 1 le bit *ALLOC_HEAP_MEMORY*. Cette initialisation est réalisée à l'aide

4. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

de l'instrumentation de la fonction *malloc* dans la bibliothèque standard C permettant d'envoyer au moniteur l'adresse et la taille de la zone mémoire allouée, et un annotation est générée au même endroit pour que le moniteur de sécurité se mette en attente de lire l'adresse et la taille de la zone mémoire à étiqueter. En effet, la fonction *malloc* utilisée pour allouer de la mémoire sur le tas ne produit pas toujours un appel système car de la mémoire préallouée est utilisée par la fonction *malloc* pour éviter le surcoût engendré par un appel système. Le même procédé est réalisé pour la fonction *free* avec cette fois l'initialisation du bit $RELEASED\_HEAP\_MEMORY$. Ces annotations ont été ajoutées à la main directement dans le code des fonctions *malloc* et *free* dans la bibliothèque standard C et permettent d'exécuter le pseudo-code affiché à droite dans le moniteur de sécurité.

Lors de l'exécution de la fonction *malloc*, le moniteur de sécurité initialise le bit $ALLOC\_HEAP\_MEMORY$ à 1 pour indiquer que la zone mémoire vient d'être allouée, et met le bit $RELEASED\_HEAP\_MEMORY$ à 0.

```
p = malloc(size);
```

$$\underset{SYSTEM}{\underline{p}}\ [ALLOC\_HEAP\_MEMORY] = 1$$
$$\dots$$
$$\underset{SYSTEM}{\underline{p+size}}\ [ALLOC\_HEAP\_MEMORY] = 1$$

$$\underset{SYSTEM}{\underline{p}}\ [RELEASED\_HEAP\_MEMORY] = 0$$
$$\dots$$
$$\underset{SYSTEM}{\underline{p+size}}\ [RELEASED\_HEAP\_MEMORY] = 0$$

Pour chaque utilisation d'une adresse mémoire, le moniteur de sécurité vérifie toujours que le bit $RELEASED\_HEAP\_MEMORY$ n'a pas été mis à 1 et donc n'a jamais été libéré. Ainsi, le moniteur de sécurité exécute pour chaque opération mémoire, le pseudo de droite qui permet la vérification que la zone mémoire est toujours valide. Cette vérification se fait à l'aide d'une annotation spéciale.

```
str r3, [r5]
ldr r2, [r5]
```

$$
\begin{aligned}
&if(\underset{SYSTEM}{address(value(r5))}[RELEASED\_HEAP\_MEMORY]\ OR\\
&\quad \dots\\
&\quad \underset{SYSTEM}{address(value(r5)+size)}[RELEASED\_HEAP\_MEMORY])\\
&\{\\
&\quad raise\_interrupt\_policy\_violation()\\
&\}
\end{aligned}
$$

Lors de la libération d'une zone mémoire avec la fonction *free*, on vérifie d'abord que la zone mémoire mémoire n'a pas déjà été libérée (RELEASED_HEAP_MEMORY).

```
free(p);
```

$$
\begin{aligned}
&if(\underset{SYSTEM}{p}[RELEASED\_HEAP\_MEMORY] == 0\\
&\quad \dots\\
&\quad \underset{SYSTEM}{p+size}[RELEASED\_HEAP\_MEMORY] == 0)\\
&\{\\
&\quad \underset{SYSTEM}{p}[RELEASED\_HEAP\_MEMORY] == 1\\
&\quad \dots\\
&\quad \underset{SYSTEM}{p+size}[RELEASED\_HEAP\_MEMORY] == 1\\
&\quad \underset{SYSTEM}{p}[ALLOC\_HEAP\_MEMORY] == 0\\
&\quad \dots\\
&\quad \underset{SYSTEM}{p+size}[ALLOC\_HEAP\_MEMORY] == 0\\
&\}\\
&else\{\\
&\quad raise\_interrupt\_policy\_violation()\\
&\}
\end{aligned}
$$

Pour pouvoir vérifier que le moniteur de sécurité détecte bien l'exploitation des vulnérabilités liées aux erreurs mémoires temporelles, on exécute d'abord le programme en utilisant une adresse mémoire qui a précédemment été libérée illustrée par le Code 3.12. Lors de l'appel à la fonction *malloc* à la ligne 14 du Code 3.11, la fonction *malloc* envoie au moniteur de sécurité les informations relatives à la zone mémoire allouée à l'adresse *0xb6efe000* avec une taille de *266240* octets.

L'appel de la fonction *free* à la ligne 16 du Code 3.11, envoie au moniteur de sécurité les informations relatives à la demande de libération de la zone

mémoire ayant l'adresse *0xb6efe000* avec une taille de *266240* octets.

Ainsi lors de la tentative d'utilisation d'une adresse précédemment libérée (ligne 5 du Code 3.11), qui est donc étiquetée avec le bit *RELEASED_HEAP_MEMORY* mis à 1, une interruption est générée par le moniteur pour avertir le système d'exploitation que l'adresse mémoire utilisée a été précédemment libérée et n'est donc plus valide.

```
0  $ ./mallocFree.elf
1      Sending Alloc Memory Request (malloc) to Monitor
2        *buff: 0xb6efe000
3        size: 266240 bytes
4
5      Sending Dealloc Memory Request (free) to Monitor
6        *buff: 0xb6efe000
7        size: 266240 bytes
8
9      [HARDBLARE Driver] pl_to_ps_irq_handler:
10       PLtoPS_Dispatcher_Status = 1
11       PLtoPS_Dispatcher_DataSize = 3
12       PLtoPS_Dispatcher_Data = 0x00010328
13       PLtoPS_Dispatcher_Data = 0xb6efe000
14       Reason: 0xb6efe000 == RELEASED_HEAP_MEMORY
```

CODE SOURCE 3.12 – Réalisation d'un *use-after-free*

Lors d'une double demande de libération d'une zone mémoire (ligne 9 du Code 3.11), le moniteur de sécurité génère également une interruption car le bit *RELEASED_HEAP_MEMORY* est déjà à 1.

```
0  $ ./mallocFree.elf 1
1
2    Sending Alloc Memory Request (malloc) to Monitor
3      *buff: 0xb6f64000
4      size: 266240 bytes
5
6    Sending Dealloc Memory Request (free) to Monitor
7      *buff: 0xb6f64000
8      size: 266240 bytes
```

```
9
10  Sending Dealloc Memory Request (free) to Monitor
11    *buff: 0xb6f64000
12    size: 266240 bytes
13
14  [HARDBLARE Driver] pl_to_ps_irq_handler:
15    PLtoPS_Dispatcher_Status = 1
16    PLtoPS_Dispatcher_DataSize = 3
17    PLtoPS_Dispatcher_Data = 0x00010328
18    PLtoPS_Dispatcher_Data = 0xb6f64000
19    Reason: 0xb6efe000 == RELEASED_HEAP_MEMORY
```

CODE SOURCE 3.13 – Réalisation d'un *double free*

Cette politique de sécurité permet ainsi de détecter les erreurs mémoires temporelles sur le tas lors de l'exécution d'une application. En plus de détecter l'exploitation de ce type de vulnérabilité, le moniteur de sécurité fournit également des informations sur le bloc de base responsable de la violation de la politique de sécurité ainsi que l'adresse mémoire concernée.

3.3.3 Politique de flux d'information pour les données provenant d'un canal de défiance

Les applications peuvent parfois avoir besoin de stocker en mémoire des données de contrôle et des données provenant de l'utilisateur. La combinaison de ces deux origines est parfois nécessaire pour, par exemple, spécifier une requête SQL qui sera envoyée au serveur de base de données. Dans ce cas, le problème que l'on souhaite éviter est le suivant : *Comment s'assurer qu'aucune commande SQL ne provienne de l'utilisateur ?* En effet, si l'utilisateur réussit à injecter de façons malintentionnées des commandes SQL grâce aux données qu'il doit fournir à l'application, une compromission de l'intégrité ou de la confidentialité de la base de données peut alors se produire. On parle alors d'injection SQL. L'une des solutions apportées avec le suivi de flux d'informations est de réserver un bit qui permettra d'étiqueter chaque donnée provenant de l'entrée standard ou de tout canal pour lequel d'administrateur système considère qu'il doit se méfier.

Ainsi lors de l'exécution de l'application, toute donnée provenant de l'utilisateur est alors marquée. Avant l'envoi de la requête SQL, l'application peut

alors vérifier qu'aucun mot-clé utilisé pour réaliser des commandes SQL ne provient de l'utilisateur.

Cette politique de flux, contrairement aux autres, est complètement gérée par le développeur de l'application. Ce n'est donc plus le moniteur de sécurité qui détecte la violation de politique de lui-même, c'est le développeur qui en fait explicitement la vérification.

Le Code 3.14 donne un exemple d'application qui permet de créer une requête SQL afin de récupérer toutes les informations présentes dans la base de données concernant l'utilisateur *Bob*, et ce uniquement si le mot de passe fourni par l'attaquant est bien celui de *Bob*. C'est donc une requête très utilisée pour pouvoir identifier un utilisateur. Puisque le mot de passe fourni ne provient pas d'un canal de confiance, toute la chaîne de caractères entrée par l'attaquant à la ligne 30 sera marquée avec le bit *SUSPICIOUS_INPUT* mis à 1. Puisqu'il s'agit d'une attaque de haut niveau, seule l'application connaît la sémantique des commandes SQL. Il est donc nécessaire d'intégrer des vérifications sur la provenance des données directement dans le code de l'application. La macro CHECK_NOT_FROM_SUSPICIOUS_INPUT définie dans le fichier *hardblare.h*, qui est utilisée dans les lignes 8/11/14 et 17, permet ainsi d'interroger le moniteur de sécurité concernant l'étiquette de l'adresse de chaque mot dans la requête SQL. Pour cela, la macro va alors instrumenter le code de l'application afin d'envoyer l'adresse de la zone mémoire ainsi que sa taille. De l'autre côté, la macro génère une annotation qui permet de lire les données envoyées par l'instrumentation et de mener une vérification du bit *SUSPICIOUS_INPUT* de la zone mémoire. Une injection SQL est donc détectée par le code de l'application, si un mot-clé présent dans la requête SQL provient d'un canal de défiance.

```
0  #include <stdio.h>
1  #include <stdlib.h>
2  #include <string.h>
3  #include <hardblare.h>
4
5  int check_integrity_word(char *sql_word)
6  {
7    if(strncmp("SELECT", sql_word, strlen("SELECT"))) {
8      CHECK_NOT_FROM_SUSPICIOUS_INPUT(sql_word, strlen(sql_word));
9    }
10   else if(strncmp("CREATE", sql_word, strlen("CREATE"))){
11     CHECK_NOT_FROM_SUSPICIOUS_INPUT(sql_word, strlen(sql_word));
12   }
13   else if(strncmp("OR", sql_word, strlen("OR"))){
14     CHECK_NOT_FROM_SUSPICIOUS_INPUT(sql_word, strlen(sql_word));
15   }
16   else if(strncmp("=", sql_word, strlen("="))){
```

```
17      CHECK_NOT_FROM_SUSPICIOUS_INPUT(sql_word, strlen(sql_word));
18    }
19    ...
20  }
21
22
23  int main(int argc, const char* argv[])
24  {
25    char input[100];
26    char sql_query[300];
27    char* word;
28    const char* separator = " ";
29
30    fgets(input,100,stdin);
31    sprintf(sql_query, "SELECT * FROM users WHERE name = 'Bob' AND
          password = '%s'", input);
32    printf("%s\n", sql_query);
33
34    word = strtok(sql_query, separator);
35
36    while (word != NULL) {
37      printf("%s\n", word);
38      check_integrity_word(word);
39      word = strtok(NULL, separator);
40    }
41    ...
42    mysql_query(con, sql_query);
43  }
```

CODE SOURCE 3.14 – Injection SQL

Lorsque l'utilisateur utilise l'application de manière attendue, et entre le mot de passe de l'utilisateur « Bob », alors la requête SQL présente à la ligne 2 du Code 3.15 est envoyée au serveur de base de données. Étant donné qu'aucun mot-clé SQL ne provient de l'utilisateur, aucune interruption n'est produite par le moniteur de sécurité.

```
0
1   $ echo -n "1234" | ./sql-injection.elf
2       SELECT * FROM users WHERE name = 'Bob' AND password = '
            ↪ 1234'
3           Check SELECT , pointer = 0xbefa2b98 , len 6
4           Check * , pointer = 0xbefa2b98 , len 1
5           Check FROM , pointer = 0xbefa2b98 , len 4
6           Check users , pointer = 0xbefa2b98 , len 5
```

```
7         Check WHERE , pointer = 0xbefa2b98 , len 5
8         Check name , pointer = 0xbefa2b98 , len 4
9         Check = , pointer = 0xbefa2b98 , len 1
10        Check 'Bob' , pointer = 0xbefa2b98 , len 7
11        Check AND , pointer = 0xbefa2b98 , len 3
12        Check password , pointer = 0xbefa2b98 , len 8
13        Check = , pointer = 0xbefa2b98 , len 1
14        Check '1234' , pointer = 0xbefa2b98 , len 6
```

CODE SOURCE 3.15 – Exécution normale de l'application avec un nom d'utilisateur

Par contre, si l'utilisateur est malintentionné, il peut alors modifier la requête SQL qui sera envoyée au serveur de base de données. En spécifiant la valeur « "0' OR '1' = '1" », l'attaquant modifie la requête et peut spécifier n'importe quel mot de passe, à savoir « 0 » dans cet exemple. La condition d'accès aux données est donc toujours vrai, peu importe si le mot de passe spécifié est bien celui de *Bob*. Le moniteur de sécurité génère alors une interruption pour informer l'administrateur système que les vérifications présentent aux lignes 13 et 16 du Code 3.14 ont échouées.

```
0   $ echo -n "0' OR '1' = '1" | ./sql-injection.elf
1       SELECT * FROM users WHERE name = 'Bob' AND password = '0'
            ↪ OR '1' = '1'
2         Check SELECT , pointer = 0xbe82bb98 , len 6
3         Check * , pointer = 0xbe82bb98 , len 1
4         Check FROM , pointer = 0xbe82bb98 , len 4
5         Check users , pointer = 0xbe82bb98 , len 5
6         Check WHERE , pointer = 0xbe82bb98 , len 5
7         Check name , pointer = 0xbe82bb98 , len 4
8         Check = , pointer = 0xbe82bb98 , len 1
9         Check 'Bob' , pointer = 0xbe82bb98 , len 7
10        Check AND , pointer = 0xbe82bb98 , len 3
11        Check password , pointer = 0xbe82bb98 , len 8
12        Check = , pointer = 0xbe82bb98 , len 1
13        Check '0' , pointer = 0xbe82bb98 , len 6
14        Check OR , pointer = 0xbe82bb98 , len 2
```

```
15
16      [HARDBLARE Driver] pl_to_ps_irq_handler:
17          PLtoPS_Dispatcher_Status = 1
18          PLtoPS_Dispatcher_DataSize = 2
19          PLtoPS_Dispatcher_Data = 0xbe82bb98
20          Reason: 0xbe82bb98 == SUSPICIOUS_INPUT
21
22      Check '1' , pointer = 0xbe82bb98 , len 3
23      Check = , pointer = 0xbe82bb98 , len 1
24
25      [HARDBLARE Driver] pl_to_ps_irq_handler:
26          PLtoPS_Dispatcher_Status = 1
27          PLtoPS_Dispatcher_DataSize = 2
28          PLtoPS_Dispatcher_Data = 0xbe82bb98
29          Reason: 0xbe82bb98 == SUSPICIOUS_INPUT
30
31      Check '1' , pointer = 0xbe82bb98 , len 3
```

CODE SOURCE 3.16 – Injection SQL

Ainsi, en proposant aux développeurs d'applications de pouvoir vérifier l'origine des données d'entrées du programme, il est possible de garantir certaines propriétés de sécurité « haut niveau ».

Ces politiques de sécurité visent à sécuriser une application. Dans la section suivante, nous détaillons comment l'administrateur système peut contrôler la propagation des données dans le système, entre différentes applications, en configurant le moniteur de sécurité et en affectant des étiquettes aux fichiers du système.

Dans la section suivante, nous évaluons l'impact de notre approche sur les performances du système.

3.4 Impacts sur les performances

La génération d'annotations et leur stockage dans le fichier exécutable influent sur la taille des fichiers exécutables. Ainsi, la taille d'un programme annoté et instrumenté uniquement dans les fichiers de l'application (sans les bibliothèques standard) est en moyenne deux fois plus volumineuse que sa

version originale. Lorsque l'on annote et instrumente également le code de la bibliothèque C (musl), le fichier exécutable devient trois fois plus volumineux. En effet, une instruction génère en moyenne une annotation, tandis que la construction de la table des blocs de base (BBT) est proportionnelle au nombre de blocs de base dans l'application. Nous rappelons que les fichiers exécutables sont liés statiquement. La figure 3.7 nous fournit la taille des fichiers exécutables issus du benchmark MiBench [5].

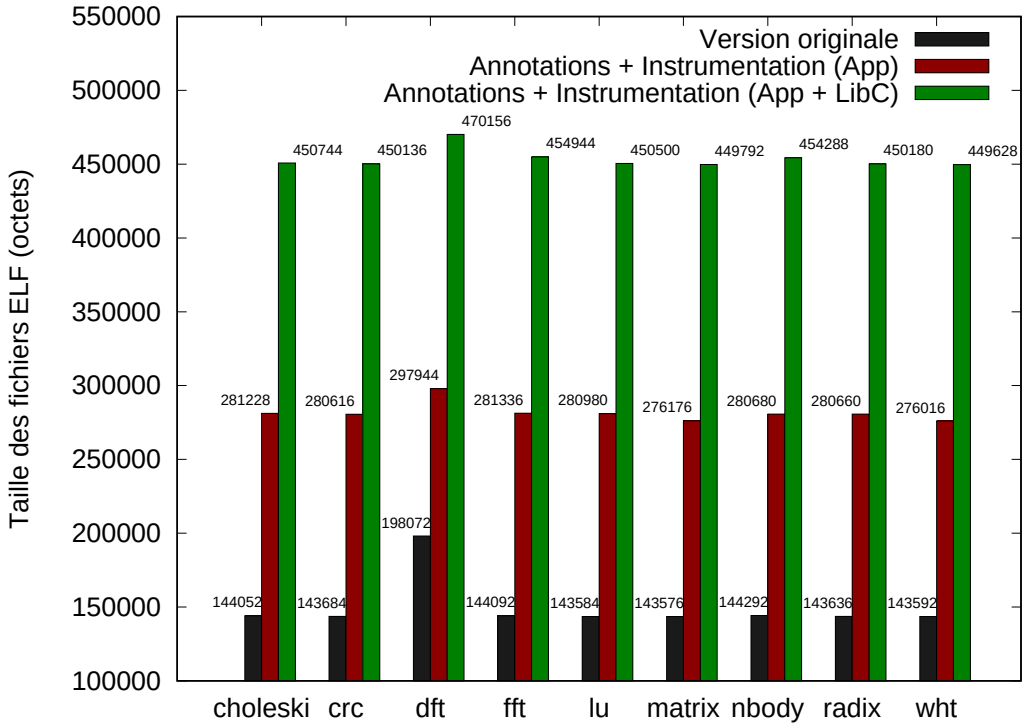


FIGURE 3.7 – Taille des fichiers exécutables

Concernant les performances, trois facteurs pénalisent le temps d'exécution de l'application monitorée sur le processeur :

1. l'envoi de données au moniteur de sécurité grâce à l'instrumentation de l'application, présenté dans la section 2.5, qui se produit uniquement lorsque les manipulations sur la mémoire ne sont pas relatives aux registres SP ou FP ;
2. la récupération d'une étiquette ou la vérification d'une étiquette par un crochet LSM ;
3. l'appel de la macro *CHECK_NOT_FROM_SUSPICIOUS_INPUT*.

5. http://vhosts.eecs.umich.edu/mibench/

Ces évènements se produisent plus ou moins fréquemment selon le type d'application. Ainsi, les applications utilisant fréquemment les appels système de lecture/écriture dans les fichiers ou celles qui accèdent fréquemment à des données en mémoire via des adresses non relatives aux registres SP ou FP, auront un surcoût plus important.

Pour avoir une estimation de l'impact sur les performances, nous avons testé différentes applications sans le contrôle de flux d'information, avec le contrôle de flux d'information uniquement dans les fichiers de l'application (sans la libC), et lorsque le contrôle de flux est également assuré dans la bibliothèque standard.

L'utilisation du contrôle de flux d'information apporte un surcoût fixe de 0,233 seconde. En effet, ce surcoût est avant tout dû à la configuration du moniteur de sécurité, de la configuration du composant de débogage et de traces, et de la mise en œuvre de l'instrumentation. Pour les applications qui ont un temps d'exécution faible, ce coût fixe devient prépondérant. Par contre, pour des applications qui ont un temps d'exécution long, tel qu'un serveur web, ce coût devient négligeable.

La figure 3.8 nous fournit les différents temps d'exécution des fichiers exécutables issus du benchmark MiBench, sans prendre en compte le coût fixe de 0,233 seconde. On peut ainsi remarquer qu'il n'y a qu'une très légère différence entre réaliser le contrôle de flux d'information dans la bibliothèque standard ou non. Ainsi, l'étendue du code supportant le contrôle de flux d'information n'a que très peu d'influence sur les performances.

Le surcoût allant jusqu'à $\times 12$ est prépondérant à cause de l'instrumentation. Une amélioration consisterait donc à réduire le plus possible les instructions d'instrumentation. Pour éviter le recours à l'instrumentation, il est possible dans certains cas de prédire la valeur d'un registre par propagation de constantes ou de prédire l'adresse pointée par un registre.

Ce chapitre a permis d'exposer les différents résultats obtenus en termes de détection de violation de politique de contrôle de flux. Nous avons donc expliqué les possibilités qu'offre le contrôle de flux d'information multicouche et multiniveau afin de garantir des propriétés de sécurité dans le système.

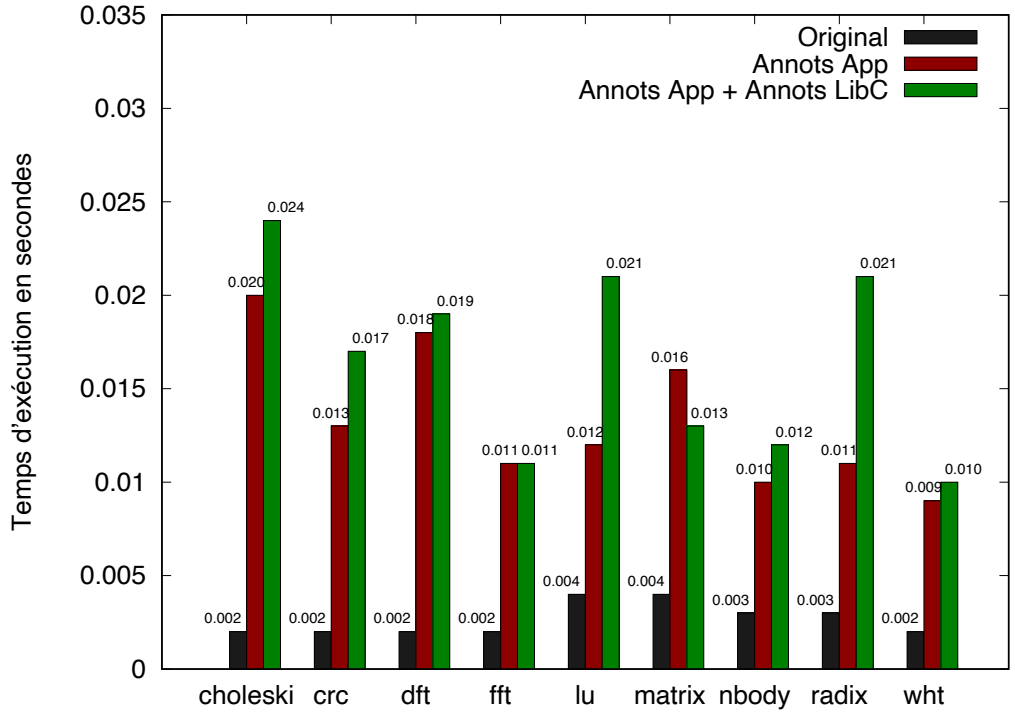


FIGURE 3.8 – Temps d'exécution dans les différentes configurations

Chapitre 4

Conclusion

Les travaux présentés dans ce manuscrit visaient à répondre à la question suivante : « **Est-il possible d'implémenter un moniteur de sécurité basé sur le suivi de flux d'information qui soit flexible, non invasif pour le matériel, qui prenne en compte les différentes couches du système, et permette de détecter différentes classes d'attaques ?** » Nous avons proposé plusieurs contributions, reposant sur des développements logiciels et matériel, permettant de répondre à cette question.

La première contribution apportée a été l'utilisation des composants de débogage ARM PTM pour réaliser du DIFT. La configuration et l'intégration de ces mécanismes a permis d'extraire des informations sur le comportement du processeur lors de l'exécution d'une application en temps réel. Puisque ce composant est une solution purement matérielle qui n'a aucune notion de l'environnement multitâche qui s'exécute sur le processeur, il a été nécessaire de filtrer la génération des traces selon le numéro d'identification du processus que l'on souhaite surveiller. Pour cela, le pilote Linux de ce composant a fait l'objet de modifications. Cette approche n'a ajouté aucun surcoût en termes de temps d'exécution de l'application.

Ce composant de traces ne fournit qu'une information partielle, à savoir les adresses cibles des branchements et celles correspondant à la survenue d'exceptions. Pour pouvoir connaître les flux d'informations qui se produisent lors de l'exécution des instructions par le processeur entre deux traces PTM, une analyse statique a été développée permettant de générer des annotations. Ces annotations propagent les étiquettes concernées dans le moniteur de sécurité. Cette analyse statique a été développée dans la partie *back-end* du compilateur pour être au plus près des instructions machine. L'analyse statique comporte tout de même une limite : les adresses mémoires manipulées par l'application qui sont affectées de façon dynamique ne peuvent être connues. Pour résoudre ce problème, une instrumentation du code permet

d'envoyer ces adresses aux moniteurs de sécurité.

Lorsque l'application surveillée réalise des appels système, le moniteur de sécurité a besoin de connaître les flux d'information engendrés. Par exemple, lors de l'écriture vers un fichier depuis une zone mémoire, il est nécessaire de propager l'étiquette de la zone mémoire vers l'étiquette du fichier. Pour ce faire, nous avons proposé un mécanisme de coopération du moniteur DIFT avec l'OS. Nous avons implémenté cette approche en modifiant le *framework* RfBlare [49].

En résumé, l'approche que nous avons proposée permet bien d'implémenter un moniteur DIFT qui ne nécessite pas de modifier le CPU. Cette approche est flexible, car différentes politiques peuvent être configurées par l'administrateur, permettant de détecter un large spectre d'attaques de natures différentes. Cette approche permet de suivre les flux d'information dans les différentes couches du système, des registres du microprocesseur aux fichiers.

Dans notre approche, nous isolons le moniteur de sécurité en l'exécutant sur un co-processeur implémenté en FPGA. Toutefois, la configuration du FPGA et du mécanisme de traces doit être réalisée par le noyau de l'OS, qui doit donc être de confiance. Cependant, le noyau peut comporter des vulnérabilités. L'attaquant pourrait exploiter ces vulnérabilités afin de désactiver notre moniteur DIFT. Afin de réduire la quantité de code qui doit être de confiance, il conviendrait d'isoler les parties du noyau en charge de la configuration du FPGA et du mécanisme de trace. Par exemple, l'architecture ARMv7 incorpore TrustZone, une technologie de sécurité permettant d'isoler une partie du code exécuté sur le processeur. Une amélioration possible de nos travaux est l'isolation complète du moniteur de sécurité, permettant ainsi de le protéger quand bien même le système d'exploitation venait à être corrompu.

La deuxième amélioration possible est le support d'un environnement multicœurs. Bien que l'architecture utilisée dans cette thèse comporte deux cœurs ARM Cortex-A9, nous n'utilisons qu'un seul cœur, ce qui nous permet de nous affranchir des éventuels problèmes liés à l'ordonnancement des différents fils d'exécution (*threads*) du processus sur les multiples cœurs du système.

Enfin, cette approche pourrait également être adaptée à d'autres architectures matérielles du marché. En effet, d'autres architectures proposent également des composants de débogage et de traces similaires à ARM PTM, par exemple Intel Processor Trace ou les mécanismes spécifiés dans la RISC-V Processor Trace Specification.

Bibliographie

[1] Laurent Georget et al. « Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory ». In : *15th International Conference on Software Engineering and Formal Methods (SEFM 2017)*. Sous la dir. d'Alessandro Cimatti et Marjan Sirjani. LNCS. Springer International Publishing, sept. 2017. doi : 10.1007/978-3-319-66197-1_1. url : http://hal.upmc.fr/hal-01535949.

[2] Nickolai Zeldovich. « Securing Untrustworthy Software Using Information Flow Control ». AAI3292438. Thèse de doct. Stanford, CA, USA, 2008. isbn : 978-0-549-35732-2.

[3] G. Edward Suh et al. « Secure Program Execution via Dynamic Information Flow Tracking ». In : *SIGARCH Comput. Archit. News* 32.5 (oct. 2004), p. 85-96. issn : 0163-5964.

[4] H. Kannan, M. Dalton et C. Kozyrakis. « Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor ». In : *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. Juin 2009. doi : 10.1109/DSN.2009.5270347.

[5] Ricardo Medel, Adriana Compagnoni et Eduardo Bonelli. « A Typed Assembly Language for Non-interference ». In : *Theoretical Computer Science*. Sous la dir. de Mario Coppo, Elena Lodi et G. Michele Pinna. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, p. 360-374. isbn : 978-3-540-32024-1.

[6] James Newsome et Dawn Song. « Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software ». In : (2005).

[7] Feng Qin et al. « LIFT : A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks ». In : *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006), p. 135-148.

[8] M. Navaki Arefi et al. « FAROS : Illuminating In-memory Injection Attacks via Provenance-Based Whole-System Dynamic Information Flow Tracking ». In : *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Juin 2018, p. 231-242. DOI : 10.1109/DSN.2018.00034.

[9] Shimin Chen et al. « Flexible Hardware Acceleration for Instruction-Grain Program Monitoring ». In : *SIGARCH Comput. Archit. News* (juin 2008). ISSN : 0163-5964. DOI : 10.1145/1394608.1382153.

[10] Robert C. Daley et Peter G. Neumann. « A General-Purpose File System for Secondary Storage ». In : *Classic Operating Systems : From Batch Processing To Distributed Systems*. Sous la dir. de Per Brinch Hansen. New York, NY : Springer New York, 2001, p. 138-166. ISBN : 978-1-4757-3510-9. DOI : 10.1007/978-1-4757-3510-9_9. URL : https://doi.org/10.1007/978-1-4757-3510-9_9.

[11] E. D. Bell et J. L. La Padula. *Secure computer system : Unified exposition and Multics interpretation*. Bedford, MA, 1976. URL : http://csrc.nist.gov/publications/history/bell76.pdf.

[12] Butler W. Lampson. « Protection ». In : *ACM SIGOPS Oper. Syst. Rev.* 8.1 (1974), p. 18-24. DOI : 10.1145/775265.775268. URL : https://doi.org/10.1145/775265.775268.

[13] D.E. Bell et L.J.L. Padula. *Secure Computer Systems : Mathematical Foundations and Model*. vol. 1. Mitre Corporation, 1973.

[14] K. J. Biba. *Integrity Considerations for Secure Computer Systems*. Rapp. tech. The Mitre Corporation, 1975.

[15] J. S. Fenton. « Memoryless subsystems ». In : *The Computer Journal* 17.2 (jan. 1974), p. 143-147. ISSN : 0010-4620. DOI : 10.1093/comjnl/17.2.143. eprint : http://oup.prod.sis.lan/comjnl/article-pdf/17/2/143/1405869/170143.pdf. URL : https://doi.org/10.1093/comjnl/17.2.143.

[16] J. A. Goguen et J. Meseguer. « Security Policies and Security Models ». In : *1982 IEEE Symposium on Security and Privacy*. Avr. 1982, p. 11-11. DOI : 10.1109/SP.1982.10014.

[17] Andrei Sabelfeld et David Sands. « Declassification : Dimensions and Principles ». In : *J. Comput. Secur.* 17.5 (oct. 2009), p. 517-548. ISSN : 0926-227X.

[18] Jia CHEN, Yu FENG et Isil DILLIG. « Precise Detection of Side-Channel Vulnerabilities Using Quantitative Cartesian Hoare Logic ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA : Association for Computing Machinery, 2017, p. 875-890. ISBN : 9781450349468. DOI : 10.1145/3133956.3134058. URL : https://doi.org/10.1145/3133956.3134058.

[19] Vineeth KASHYAP, Ben WIEDERMANN et Ben HARDEKOPF. « Timing- and Termination-Sensitive Secure Information Flow : Exploring a New Approach ». In : *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. SP '11. USA : IEEE Computer Society, 2011, p. 413-428. ISBN : 9780769544021. DOI : 10.1109/SP.2011.19. URL : https://doi.org/10.1109/SP.2011.19.

[20] O. ZIBORDI DE PAIVA et W. V. RUGGIERO. « A survey on Information Flow Control mechanisms in web applications ». In : *2015 International Conference on High Performance Computing Simulation (HPCS)*. 2015, p. 211-220.

[21] Michael DALTON, Hari KANNAN et Christos KOZYRAKIS. « Raksha : A Flexible Information Flow Architecture for Software Security ». In : *SIGARCH Comput. Archit. News* 35.2 (juin 2007), p. 482-493. ISSN : 0163-5964.

[22] *2020 CWE Top 25 Most Dangerous Software Weaknesses*. MITRE. 2020. URL : https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html (visité le 14/09/2020).

[23] Nevin HEINTZE et Jon G. RIECKE. « The SLam Calculus : Programming with Secrecy and Integrity ». In : *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA : ACM, 1998, p. 365-377. ISBN : 0-89791-979-3. DOI : 10.1145/268946.268976. URL : http://doi.acm.org/10.1145/268946.268976.

[24] Dennis VOLPANO et Geoffrey SMITH. « A type-based approach to program security ». In : *TAPSOFT '97 : Theory and Practice of Software Development*. Sous la dir. de Michel BIDOIT et Max DAUCHET. Berlin, Heidelberg : Springer Berlin Heidelberg, 1997, p. 607-621. ISBN : 978-3-540-68517-3.

[25] *Perl Security*. URL : https://perldoc.perl.org/perlsec.html.

[26] Dorothy E. Denning. « A Lattice Model of Secure Information Flow ». In : *Commun. ACM* 19.5 (mai 1976), p. 236-243. ISSN : 0001-0782. DOI : 10.1145/360051.360056. URL : http://doi.acm.org/10.1145/360051.360056.

[27] Butler W. Lampson. « A Note on the Confinement Problem ». In : *Commun. ACM* 16.10 (oct. 1973), p. 613-615. ISSN : 0001-0782. DOI : 10.1145/362375.362389. URL : http://doi.acm.org/10.1145/362375.362389.

[28] Marvin L. Minsky. *Computation : Finite and Infinite Machines*. USA : Prentice-Hall, Inc., 1967. ISBN : 0131655639.

[29] A. Sabelfeld et A. C. Myers. « Language-based Information-flow Security ». In : *IEEE J.Sel. A. Commun.* 21.1 (sept. 2006), p. 5-19. ISSN : 0733-8716. DOI : 10.1109/JSAC.2002.806121. URL : https://doi.org/10.1109/JSAC.2002.806121.

[30] Henry Gordon Rice. « Classes of recursively enumerable sets and their decision problems ». In : *Transactions of the American Mathematical Society* 74.2 (1953), p. 358-366.

[31] Bernhard Reus. *Limits of Computation*. Springer, 2016.

[32] Dorothy E. Denning et Peter J. Denning. « Certification of Programs for Secure Information Flow ». In : *Commun. ACM* 20.7 (juill. 1977), p. 504-513. ISSN : 0001-0782.

[33] Andrew C. Myers et Barbara Liskov. « Protecting Privacy Using the Decentralized Label Model ». In : *ACM Trans. Softw. Eng. Methodol.* 9.4 (oct. 2000), p. 410-442. ISSN : 1049-331X. DOI : 10.1145/363516.363526. URL : http://doi.acm.org/10.1145/363516.363526.

[34] François Pottier et Vincent Simonet. « Information Flow Inference for ML ». In : *ACM Trans. Program. Lang. Syst.* 25.1 (jan. 2003), p. 117-158. ISSN : 0164-0925. DOI : 10.1145/596980.596983. URL : http://doi.acm.org/10.1145/596980.596983.

[35] Gilles Barthe, David Pichardie et Tamara Rezk. « A Certified Lightweight Non-interference Java Bytecode Verifier ». In : *Programming Languages and Systems*. Sous la dir. de Rocco De Nicola. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 125-140. ISBN : 978-3-540-71316-6.

[36] Dennis Volpano, Cynthia Irvine et Geoffrey Smith. « A Sound Type System for Secure Flow Analysis ». In : *J. Comput. Secur.* 4.2-3 (jan. 1996), p. 167-187. ISSN : 0926-227X. URL : http://dl.acm.org/citation.cfm?id=353629.353648.

[37] Andrew C. Myers et Andrew C. Myers. « JFlow : Practical Mostly-static Information Flow Control ». In : *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA : ACM, 1999, p. 228-241. ISBN : 1-58113-095-3. DOI : 10.1145/292540.292561. URL : http://doi.acm.org/10.1145/292540.292561.

[38] François Pottier et Sylvain Conchon. « Information Flow Inference for Free ». In : *SIGPLAN Not.* 35.9 (sept. 2000), p. 46-57. ISSN : 0362-1340. DOI : 10.1145/357766.351245. URL : http://doi.acm.org/10.1145/357766.351245.

[39] Fabrice Bellard. « QEMU, a Fast and Portable Dynamic Translator ». In : *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA : USENIX Association, 2005, p. 41-41. URL : http://dl.acm.org/citation.cfm?id=1247360.1247401.

[40] Christoph Kerschbaumer et al. « Information Flow Tracking Meets Just-in-Time Compilation ». In : *ACM Trans. Archit. Code Optim.* 10.4 (déc. 2013). ISSN : 1544-3566. DOI : 10.1145/2541228.2555295. URL : https://doi.org/10.1145/2541228.2555295.

[41] Andrew C. Myers, Andrew C. Myers et Barbara Liskov. « A Decentralized Model for Information Flow Control ». In : *SIGOPS Oper. Syst. Rev.* 31.5 (oct. 1997), p. 129-142. ISSN : 0163-5980. DOI : 10.1145/269005.266669. URL : http://doi.acm.org/10.1145/269005.266669.

[42] Taegyu Kim et al. « RevARM : A Platform-Agnostic ARM Binary Rewriter for Security Applications ». In : *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC 2017. Orlando, FL, USA : ACM, 2017, p. 412-424. ISBN : 978-1-4503-5345-8. DOI : 10.1145/3134600.3134627. URL : http://doi.acm.org/10.1145/3134600.3134627.

[43] Vasanth Bala, Evelyn Duesterwald et Sanjeev Banerjia. « Dynamo : A Transparent Dynamic Optimization System ». In : *SIGPLAN Not.* 46.4 (mai 2011), p. 41-52. ISSN : 0362-1340. DOI : 10.1145/1988042.1988044. URL : http://doi.acm.org/10.1145/1988042.1988044.

[44] Gregory T. Sullivan et al. « Dynamic Native Optimization of Interpreters ». In : *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. IVME '03. San Diego, California : ACM, 2003, p. 50-57. ISBN : 1-58113-655-2. DOI : 10.1145/858570.858576. URL : http://doi.acm.org/10.1145/858570.858576.

[45] Chi-Keung Luk et al. « Pin : Building Customized Program Analysis Tools with Dynamic Instrumentation ». In : *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA : ACM, 2005, p. 190-200. ISBN : 1-59593-056-6. DOI : 10.1145/1065010.1065034. URL : http://doi.acm.org/10.1145/1065010.1065034.

[46] Nicholas Nethercote et Julian Seward. « Valgrind : A Framework for Heavyweight Dynamic Binary Instrumentation ». In : *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA : ACM, 2007, p. 89-100. ISBN : 978-1-59593-633-2. DOI : 10.1145/1250734.1250746. URL : http://doi.acm.org/10.1145/1250734.1250746.

[47] Petros Efstathopoulos et al. « Labels and Event Processes in the Asbestos Operating System ». In : *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom : Association for Computing Machinery, 2005, p. 17-30. ISBN : 1595930795. DOI : 10.1145/1095810.1095813. URL : https://doi.org/10.1145/1095810.1095813.

[48] Indrajit Roy et al. « Laminar : Practical Fine-grained Decentralized Information Flow Control ». In : *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland : ACM, 2009, p. 63-74. ISBN : 978-1-60558-392-1. DOI : 10.1145/1542476.1542484. URL : http://doi.acm.org/10.1145/1542476.1542484.

[49] Laurent Georget et al. « Verifying the Reliability of Operating System-Level Information Flow Control Systems in Linux ». In : *5th International FME Workshop on Formal Methods in Software Engineering*. Buenos Aires, Argentina : IEEE Press, mai 2017, p. 10-16. DOI : 10.1109/FormaliSE.2017.1. URL : https://hal.inria.fr/hal-01535862.

[50] Laurent Georget et al. « Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory ». In : *15th International Conference on Software Engineering and Formal Methods (SEFM 2017)*. Sous la dir. d'Alessandro Cimatti et Marjan Sirjani. LNCS. Trento, Italy : Springer International Publishing, sept. 2017, p. 1-16. DOI : 10.1007/978-3-319-66197-1_1. URL : https://hal.sorbonne-universite.fr/hal-01535949.

[51] C. Wright et al. « Linux security modules : general security support for the linux kernel ». In : *Foundations of Intrusion Tolerant Systems,*

2003 [Organically Assured and Survivable Information Systems]. Déc. 2003, p. 213-226. DOI : 10.1109/FITS.2003.1264934.

[52] Adwait NADKARNI et al. « Practical DIFC Enforcement on Android ». In : *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA : USENIX Association, 2016, p. 1119-1136. ISBN : 978-1-931971-32-4. URL : http://dl.acm.org/citation.cfm?id=3241094.3241181.

[53] Valérie VIET TRIEM TONG, Andrew CLARK et Ludovic MÉ. « Specifying and enforcing a fine-grained information flow policy : model and experiments ». In : *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications* 1.1 (juin 2010). This paper was part of the 2nd International Workshop on Managing Insider Security Threats (MIST 2010) ; Morika, Iwate, Japan (14-15 June 2010)., p. 56-71. URL : https://hal-supelec.archives-ouvertes.fr/hal-00516672.

[54] Daniel TOWNLEY et al. « LATCH : A Locality-Aware Taint CHecker ». In : (2019).

[55] Olatunji RUWASE et al. « Parallelizing Dynamic Information Flow Tracking ». In : *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08. Munich, Germany : ACM, 2008, p. 35-45. ISBN : 978-1-59593-973-9.

[56] Shimin CHEN et al. « Log-Based Architectures for General-Purpose Monitoring of Deployed Code ». In : *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. ASID '06. San Jose, California : Association for Computing Machinery, 2006, p. 63-65. ISBN : 1595935762. DOI : 10.1145/1181309.1181319. URL : https://doi.org/10.1145/1181309.1181319.

[57] Kangkook JEE et al. « ShadowReplica : Efficient Parallelization of Dynamic Data Flow Tracking ». In : *Proceedings of CCS*. Berlin, Germany : ACM, 2013, p. 235-246. ISBN : 978-1-4503-2477-9.

[58] G. VENKATARAMANI et al. « FlexiTaint : A programmable accelerator for dynamic taint propagation ». In : *2008 IEEE 14th HPCA*. Fév. 2008, p. 173-184. DOI : 10.1109/HPCA.2008.4658637.

[59] T. M. AUSTIN. « DIVA : a reliable substrate for deep submicron microarchitecture design ». In : *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. Nov. 1999, p. 196-207. DOI : 10.1109/MICRO.1999.809458.

[60] Neetu JINDAL et al. « DHOOM : Reusing Design-for-Debug Hardware for Online Monitoring ». In : *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA : ACM, 2019, 99 :1-99 :6. ISBN : 978-1-4503-6725-7. DOI : 10.1145/3316781.3317799. URL : http://doi.acm.org/10.1145/3316781.3317799.

[61] Normann DECKER et al. « Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems ». In : *Formal Methods : Foundations and Applications*. Sous la dir. de Simone CAVALHEIRO et José FIADEIRO. Cham : Springer International Publishing, 2017, p. 179-196. ISBN : 978-3-319-70848-5.

[62] S. MOORE et S. CHONG. « Static Analysis for Efficient Hybrid Information-Flow Control ». In : *2011 IEEE 24th Computer Security Foundations Symposium*. Juin 2011, p. 146-160. DOI : 10.1109/CSF.2011.17.

[63] A. RUSSO et A. SABELFELD. « Dynamic vs. Static Flow-Sensitive Security Analysis ». In : *2010 23rd IEEE Computer Security Foundations Symposium*. Juill. 2010, p. 186-199. DOI : 10.1109/CSF.2010.20.

[64] B. JAIN et al. « SoK : Introspections on Trust and the Semantic Gap ». In : *2014 IEEE Symposium on Security and Privacy*. 2014, p. 605-620.

[65] ARM. *CoreSight Components - Technical Reference manual*. 2009.

[66] James REINDERS. *Processor Tracing*. 2013. URL : https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[67] Gajinder PANESAR et Iain ROBERTSON. *RISC-V Processor Trace*. URL : https://github.com/riscv/riscv-trace-spec/blob/master/riscv-trace-spec.pdf.

[68] *Embedded Trace Macrocell Architecture Specification*. 2011. URL : https://developer.arm.com/documentation/ihi0014/q.

[69] *CoreSight Program Flow Trace - Architecture Specification*. 2011. URL : https://developer.arm.com/documentation/ihi0035/b.

[70] Muhammad Abdul WAHAB. « Hardware support for the security analysis of embedded softwares : applications on information flow control and malware analysis ». 2018CSUP0003. Thèse de doct. 2018. URL : http://www.theses.fr/2018CSUP0003/document.

[71] Swarup BHUNIA et Mark M. TEHRANIPOOR. *The Hardware Trojan War : Attacks, Myths, and Defenses*. 1st. Springer Publishing Company, Incorporated, 2017. ISBN : 3319685104.

[72] Leonel Acunha guimarães. « Testing Techniques for Detection of Hardware Trojans in Integrated Circuits of Trusted Systems ». Theses. Université Grenoble Alpes, déc. 2017. URL : https://tel.archives-ouvertes.fr/tel-01754790.

[73] R. S. Chakraborty et S. Bhunia. « Security against hardware Trojan through a novel application of design obfuscation ». In : *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. 2009, p. 113-116.

[74] Tobias Schneider, Amir Moradi et Tim Güneysu. « ParTI – Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks ». In : août 2016, p. 302-332. ISBN : 978-3-662-53007-8. DOI : 10.1007/978-3-662-53008-5_11.

[75] William D. Casper et Stephen M. Papa. « Trusted Boot ». In : *Encyclopedia of Cryptography and Security*. Sous la dir. d'Henk C. A. van Tilborg et Sushil Jajodia. Boston, MA : Springer US, 2011, p. 1327-1328. ISBN : 978-1-4419-5906-5. DOI : 10.1007/978-1-4419-5906-5_794. URL : https://doi.org/10.1007/978-1-4419-5906-5_794.

[76] Muhammad Abdul Wahab. « Hardware support for the security analysis of embedded softwares : applications on information flow control and malware analysis ». Theses. CentraleSupélec, déc. 2018. URL : https://tel.archives-ouvertes.fr/tel-02634340.

[77] H. Cheng et al. « Trading Conditional Execution for More Registers on ARM Processors ». In : *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. 2010, p. 53-59.

[78] Hans-Jürgen Koch. *The Userspace I/O HOWTO*. 2006. URL : https://www.kernel.org/doc/html/v4.13/driver-api/uio-howto.html.

[79] Xinyang Ge, Weidong Cui et Trent Jaeger. « GRIFFIN : Guarding Control Flows Using Intel Processor Trace ». In : *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China : Association for Computing Machinery, 2017, p. 585-598. ISBN : 9781450344654. DOI : 10.1145/3037697.3037716. URL : https://doi.org/10.1145/3037697.3037716.

Table des figures

Liste des tableaux

Liste des codes sources

Glossaire

ABI Application Binary Interface. 78, 101

API Application Programming Interface. 25

APSR Application Program Status Register. 49, 55, 57, 62, 63, 67, 68, 93

AXI Advanced eXtensible Interface. 39, 71, 84, 85

BBT Basic Block Table. 83, 85, 87, 103, 120

BRAM Block RAM Memory. 85, 86

CFI Control Flow Integrity. 100

CPU Central Processing Unit. xi, 11, 24, 27, 28, 30, 32, 37, 39, 44, 54, 57, 59, 60, 86, 124

DAC Discretionary access control. 5, 24

DBI Dynamic Binary Instrumentation. 21, 22, 35

DIFT Dynamic Information Flow Tracking. x–xii, 19, 21–24, 26–33, 39–41, 79, 81, 87, 91, 94, 123, 124

DoD U.S. Department of Defense. 5

ELF Executable and Linkable Format. 21, 82, 84, 86

EMIO Extended Multiplexed Input Output. 48

ETM Embedded Trace Macrocell. 31

FIFO First In, First Out. 48, 71, 72, 85, 86

FP Frame Pointer. 72, 120, 121

FPGA Field Programmable Gate Array. 28, 31, 37, 39

IFC Information Flow Control. 6, 13

In-Core In-Core. 35

ISA Instruction Set Architecture. 49, 59
itag Information tag. 25

JIT Just-In-Time. 19
JVM Java Virtual Machine. 26

LBA Log-Based Architectures. 29
Little-endian Petit-boutisme. 108
LLVM Low-Level Virtual Machine. 41, 43, 50, 87
LR Link Register. 54, 55, 101, 102
LSM Linux Security Modules. 23, 25, 26, 43, 80, 81, 120

MAC Mandatory access control. 5, 24
MITRE The MITRE Corporation. 45, 111
MMU Memory Management Unit. 58

nop No Operation. 50
NSA National Security Agency. 23, 79

Off-Core Off-Core. 35
Off-Loading Off-Loading. 35
OS Operating System. vii, x–xii, 26, 35, 48, 75, 81, 82, 90, 124

PC Program Counter. 14, 18, 44, 52, 54–57, 63, 67, 101, 102
PFT Program Flow Trace. 48
PID Process IDentifier. 48
PL Programmable Logic. 39, 48
PLT Procedure Linkage Table. 21
PS Processing System. 39, 86
ptag Policy tag. 25
PTM Program Trace Macrocell. 38, 39, 47–53, 55, 82, 83, 85, 103

RISC Reduced Instruction Set Computer. 54, 85
ROP Return Oriented Programming. 20, 22, 100

SoC System on Chip. 37
Softcore Softcore. 28, 30

SP Stack Pointer. 54, 55, 72, 73, 120, 121

TM Tag Memory. 59, 86

TPIU Trace Port Interface unit. 48

TRF Tag Register File. 57, 86

UIO Userspace I/O. 71, 75

Use After Free Use After Free. 100

VM Virtual Machine. 35

DOCTORAT BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

Titre : Contrôle de flux d'information par utilisation conjointe d'analyse statique et dynamique accélérée matériellement.

Mots clés : sécurité, contrôle de flux d'information, systèmes embarqués, composants de débogage et de traces, analyse statique, instrumentation, Linux Security Modules.

Résumé : Les systèmes embarqués étant de plus en plus présents dans nos vies, il est nécessaire de protéger les données personnelles qui y sont stockées. En effet, les concepteurs d'applications peuvent involontairement introduire des vulnérabilités pouvant être exploitées par un attaquant pour compromettre la confidentialité ou l'intégrité du système. Un des moyens de s'en prémunir est l'utilisation d'outils réactifs permettant de surveiller le comportement du système lors de son fonctionnement. Dans le cadre de cette thèse, nous proposons une approche générique de détection d'anomalies combinant des aspects matériels et logiciels et qui repose sur le suivi de flux d'information dynamique (DIFT). Le DIFT consiste à attacher des étiquettes représentant des niveaux de sécurité à des conteneurs d'information, par exemple des fichiers, et à spécifier une politique de flux d'information permettant de décrire les flux autorisés. Pour cela, nous avons tout d'abord développé un moniteur DIFT, flexible et non invasif pour le processeur, en utilisant les composants de traces ARM CoreSight. Pour prendre en compte les flux d'information qui se produisent dans les différentes couches, du système d'exploitation aux instructions processeur, nous avons élaboré des analyses statiques dans le compilateur. Ces analyses génèrent des annotations qui décrivent la dissémination des données dans le système lors de son exécution et qui sont utilisées par le moniteur DIFT. Nous avons également développé un module de sécurité pour le noyau Linux afin de prendre en compte les flux d'information à destination ou en provenance des fichiers. L'approche proposée permet ainsi de détecter un large spectre d'attaques de natures différentes.

Title : A hardware-accelerated information flow control monitor by joint use of static and dynamic analysis.

Keywords : security, information flow control, embedded systems, debug and trace components, static analysis, instrumentation, Linux Security Modules

Abstract : As embedded systems are more and more present in our lives, it is necessary to protect the personal data stored in such systems. Application developers can unintentionally introduce vulnerabilities that can be exploited by attackers to compromise the confidentiality or integrity of the system. One of the solutions to prevent this is to use reactive mechanisms to monitor the behavior of the system while it is running. In this thesis, we propose a generic anomaly detection approach combining hardware and software aspects, based on dynamic information flow tracking (DIFT). DIFT consists of attaching labels representing security levels to information containers, for example files, and specifying an information flow policy to describe the authorized flows. To implement such an approach, we first developed a DIFT monitor which is flexible and non-invasive for the processor, using ARM CoreSight trace components. To take into account the information flows that occur in the different layers, from the operating system to the processor instructions, we have developed different static analysis into the compiler. These analyses generate annotations, used by the DIFT monitor, that describe the dissemination of data in the system at run-time. We also developed a Linux security module to handle information flows involving files. The proposed approach can thus be used to detect different kinds of attacks.