Sensibilité de logiciels au détournement de flot de contrôle

Etienne Louboutin

DOCTORAT BRETAGNE LOIRE / MATHSTIC

...

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'École Nationale Supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire - IMT Atlantique

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Informatique

Par

Étienne LOUBOUTIN

Sensibilité de logiciels au détournement de flot de contrôle

Thèse présentée et soutenue à IMT Atlantique, Brest, le 11 Janvier 2021
Unité de recherche : Lab-STICC
Thèse N° : 2021IMTA0230

Rapporteurs avant soutenance :

Marie-Laure POTET	Professeur, VERIMAG
David HELY	Maître de conférence HDR, LCIS

Composition du Jury :

Président :	Caroline FONTAINE	Directrice de recherche, LSV, ENS Paris-Saclay
Examinateurs :	Erven ROUHOU	Directeur de recherche, IRISA Rennes
	Marie-Laure POTET	Professeur, VERIMAG
	David HELY	Maître de conférence HDR, LCIS
Dir. de thèse :	Fabien DAGNAT	Maître de conférences HDR, IMT Atlantique
Encadrant :	Jean-Christophe BACH	Maître de conférence, IMT Atlantique

Invité(s) :

Laurent FREREBEAU	Responsable de service logiciel de sécurité, Thalès

Table des matières

Remerciements

Avant de commencer la lecture de cette, thèse vous trouverez ici toutes les personnes que je souhaite remercier, souvent non nommées pour éviter d'en oublier et qui m'ont permis d'aller au bout des travaux, de la rédaction et des différents évènements qui ont rythmé ce doctorat.

Dans un premier temps, je voudrais remercier les rapporteurs de ma thèse et l'intégralité de mon jury pour avoir pris le temps de lire et prendre connaissance de mes travaux. Ensuite, je voudrais remercier mon directeur de thèse Fabien Dagnat et mon encadrant Jean-Christophe Bach pour le support, et toutes les discussions scientifiques ou non qu'on a eu lors de ces quatre années de thèse. Je remercie aussi mon équipe d'accueil au CEA qui m'a laissé un peu de temps pour clore ce doctorat.

Ensuite, j'aimerais remercier tous mes collègues, à la fois à IMT-Atlantique et à l'École Navale avec qui j'ai pu échanger scientifiquement autour de mes travaux. Notamment David, qui en plus de me conseiller en termes de rédaction et de compétence scientifique, a du supporter mon humour et ma présence.

Je pense aussi à tous les autres doctorants, de la chaire de cyberdéfense des systèmes navals, et ceux d'IMT Atlantique avec j'ai interagi, bu des bières scientifiquement ou non. Je remercie aussi Bastien, qui m'a permis de découvrir l'École Navale sous des angles insoupçonnés ainsi que toutes les personnes avec qui j'ai pu m'enrichir dans la magnifique ville de Brest, et ses alentours.

Enfin, je remercie ma famille qui était tout de même derrière moi à me soutenir et me pousser à finir.

Introduction

La sécurité à la conception, ou *security by design*, est une approche de la conception logicielle récente. L'idée de cette approche est de prendre en compte la sécurité dès les phases initiales de la conception d'un logiciel. La sécurité n'est plus alors une contrainte qui arrive tardivement dans le cycle de développement, mais suffisamment tôt pour influer sur l'architecture du logiciel. Cette approche influe aussi sur d'autres décisions de conception, comme les choix de compilateur, de langages ou même la cible matérielle.

Initialement, les attaques contre un logiciel étaient faciles à mettre en œuvre. Un attaquant injectait directement en mémoire un programme, souvent nommée *shell code*, et le faisait exécuter par le système. Ce type d'exploitation n'est plus vraiment possible aujourd'hui, grâce à l'introduction du DEP (*Data Execution Prevention*). Cette protection consiste à marquer, lors de l'exécution d'un programme, une zone mémoire soit exécutable soit ouverte en écriture, mais jamais ne permettre les deux. Ce type de protection est maintenant déployé sur tous les systèmes complexes, suite à un support matériel général.

Pour contourner ces protections, les attaques modernes utilisent du code légitime présent dans la mémoire du programme ciblé. Le code étant déjà marqué comme exécutable dans la mémoire, le processeur et le système ne peuvent pas faire la différence entre l'exécution légitime et l'attaque uniquement en utilisant le DEP. Les différents éléments constituant une attaque par réutilisation de code sont nommés gadgets. Leur première définition date de la description sur l'architecture x86 du *Return-Oriented Programming*. Le gadget y est défini comme une suite d'instructions assembleur, présente dans le code du programme attaqué, terminée par une instruction `return`. Depuis cette définition, d'autres instructions, qui utilisent une adresse en mémoire pour diriger l'exécution, servent de saut. La suite d'instructions qui les précèdent devient donc également des gadgets. Une telle

attaque par détournement de flot de contrôle, parfois aussi nommée réutilisation de code, chaîne ces gadgets pour parvenir à exécuter la suite d'instructions voulue. La charge utile de l'attaque est donc appelée chaîne de détournement de flot de contrôle. Les attaques par réutilisation de code nécessitent une corruption de mémoire pour démarrer le détournement d'exécution.

Une corruption mémoire est la modification illégitime d'une zone mémoire utilisée par le programme. Les attaques contre des logiciels ont souvent besoin d'une corruption mémoire pour être initiées. Un ingénieur de Microsoft [Mil19] avait annoncé que plus de 70 % des failles proviennent d'une mauvaise gestion mémoire. Plusieurs choix de conception peuvent limiter ces problèmes de mémoire. Par exemple, certains langages sont considérés comme *memory safe*, car le modèle mémoire ne laisse pas toute liberté à un développeur. L'utilisation de ce type de langage ne suffit cependant pas. En effet, des attaques par canaux auxiliaires, matériels ou logiciels, peuvent permettre la modification d'une zone mémoire considérée comme protégée par le programme en cours d'exécution.

Ainsi, indépendamment du caractère sûr de notre application, il convient de la protéger vis-à-vis des attaques par détournement de flot de contrôle. Plusieurs protections ont été proposées depuis la description académique des attaques par réutilisation de code. Ces protections ne sont pas déployées de manière généralisée pour des raisons de performances. En effet, elles dégradent fortement les performances des programmes protégés. De plus, lors de la conception d'un système, ces protections arrivent tardivement et ne peuvent pas avoir d'influence sur les choix techniques. En effet, les protections se basent sur une analyse du binaire produit ou sur la représentation interne du compilateur. Un des seuls cas où elles ne posent pas de contraintes à la conception du système est si les dépendances du programme, par exemple la `libc`, doivent aussi être protégées, ce qui n'est pas toujours envisageable.

Pour choisir la protection qui sera déployée, un développeur a besoin de mesurer l'efficacité des différentes solutions. Une protection est mise en place pour limiter ou vérifier certains comportements. Chaque protection nécessite de faire des hypothèses. Ces hypothèses portent sur les capacités d'un attaquant, comme les droits qu'il a sur la mémoire du programme. Par exemple, l'efficacité de la protection *Code Pointer Integrity* est garantie dans le cas où une partie de la mémoire du programme peut être cachée en lecture à l'attaquant. C'est une hypothèse très forte rarement faisable.

Ensuite, lorsqu'une application est conçue dans une approche de *security*

by design, il faut pouvoir comparer la sécurité induite par tous les outils utilisés pendant la production en plus des solutions de protection. Comparer la facilité d'exploitation de deux applications est difficile. D'un point de vue quantitatif, mesurer le nombre de gadgets utilisables est facile. Savoir si les gadgets contenus dans le binaire permettent de construire une attaque est cependant plus complexe. Cette quantité d'éléments peut donner des indices sur des paramètres influant sur la complexité à créer une chaîne de détournement de flot de contrôle.

Pour mettre en place des outils et des méthodes permettant d'aider le développement d'applications sécurisées contre ce type d'attaque, je me suis intéressé aux logiciels installés sur des systèmes existants pour mener des analyses. Dans un premier temps, seuls des binaires systèmes ont été analysés. J'ai différencié une application, des binaires associés. Plusieurs binaires peuvent, en effet, correspondre à une seule application. Car cette application peut être produite par différents compilateurs, sur différents systèmes et elle peut aussi avoir plusieurs versions. Toutes ces variations conduisent à des binaires différents qu'il convient de comparer.

La comparaison de différents binaires, de procédés, de langages et autres outils, nécessite des métriques. J'ai donc mis en place des métriques pour quantifier et qualifier la facilité d'exploitation d'un binaire par les attaques par détournement de flot de contrôle. J'ai ensuite validé ces métriques en les utilisant sur des écosystèmes d'applications variés. Ces métriques ont permis d'identifier certaines applications qui semblent plus facilement exploitables que d'autres.

Pour permettre de produire de meilleurs binaires, j'ai mis en relation la sensibilité d'un binaire et la manière avec laquelle il a été produit. Différents outils et paramètres de construction d'une application peuvent être considérés sur un système. Un système regroupe souvent des applications de sources bien différentes. On retrouve donc des applications ayant été écrites dans différents langages, produites avec différents compilateurs, différentes options, etc. Je me suis donc intéressé à l'étude de ces applications pour déterminer si certaines contiennent plus d'éléments utilisables pour construire des attaques que d'autres.

Ces informations servent à mettre en place des règles ou recommandations pour concevoir une application ou un système moins sensible aux attaques par détournement de flot de contrôle. Des acteurs différents ont des spécifications propres à leur métier, et ces recommandations peuvent permettre aussi de choisir des protections adéquates à chacun de ces niveaux.

Plan

Ce document est composé de 4 chapitres. Le premier chapitre met en place le contexte de la thèse, de la sécurité des logiciels et du concept de *security by design*. Il contient les éléments nécessaires à la compréhension des attaques par détournement de flot de contrôle. Il présente les différentes solutions de protection existantes. Les moyens disponibles pour évaluer l'efficacité des dites solutions y sont aussi discutés.

Le deuxième chapitre montre le procédé mis en place pour définir des métriques de sensibilité d'un logiciel vis-à-vis des attaques par détournement de flot de contrôle. L'utilisabilité des métriques en question y est discutée, et les résultats permettant d'avoir une idée de la sensibilité d'un logiciel y sont présentés.

Dans le troisième chapitre, je discute de la mise en œuvre de ces métriques pour qualifier les procédés utilisés lors de la production d'un binaire pour un logiciel. L'influence de langage, compilateur, options, et cible matérielle sur ces métriques y est présentée.

Le quatrième chapitre montre comment utiliser ces résultats lors de la conception d'un logiciel. La prise en compte des attaques par détournement de flot de contrôle dès les premières étapes de conception y est développée.

Chapitre 1

Attaques et protections des logiciels

1.1 Introduction

Ce chapitre présente les concepts utilisés dans la thèse, en commençant par définir ce que je considère comme un logiciel. Le logiciel et son exécution y sont détaillés pour décrire le fonctionnement des attaques par réutilisation de code. J'y présente aussi le contexte dans lequel les attaques contre des logiciels sont considérées, et le modèle d'attaque pris en compte. Ce chapitre présente ensuite un ensemble de protections existantes contre les attaques considérées, ainsi qu'une synthèse de l'efficacité des protections et de leurs performances.

1.2 Définitions et concepts

Dans ce document, différents termes sont utilisés. Une définition claire de ces termes est nécessaire. Le premier concept utilisé tout au long de cette thèse est celui de logiciel. Un logiciel est défini comme suit : un logiciel est un outil informatique fournissant un ou plusieurs services.

Par exemple, `Firefox` ou `tar` sont deux logiciels. Ces deux logiciels sont pris comme exemple à plusieurs endroits dans cette thèse. Les logiciels sont répartis dans deux catégories. La première catégorie est celle des logiciels utilisateurs. Cette catégorie regroupe les logiciels utilisables par eux-mêmes et qui n'ont pas vocation à être intégrés dans d'autres logiciels. Les deux exemples utilisés ci-dessus sont dans cette catégorie. La seconde catégorie est celle des bibliothèques. Ces logiciels sont utilisés pour développer d'autres logiciels, utilisateurs ou non. Par exemple,

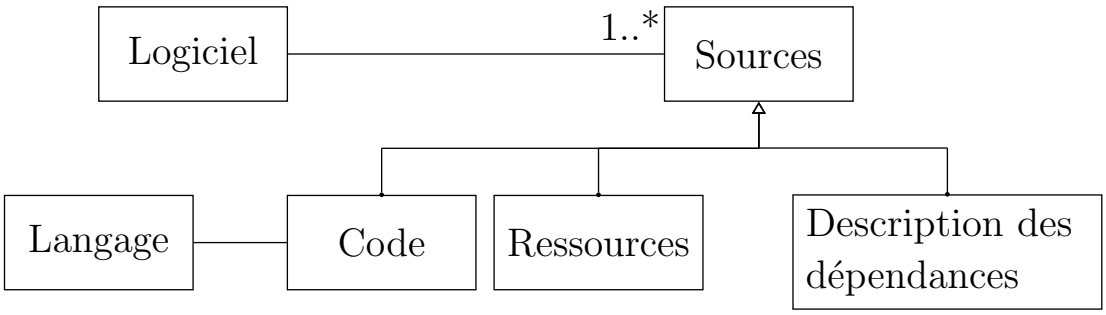


FIGURE 1.1 – Relation entre les concepts hauts niveaux autour du logiciel

on trouve dans cette catégorie les bibliothèques classiques comme la bibliothèque C standard ou `OpenGL`.

Chaque logiciel est associé aux sources qui implémentent ce logiciel. Les sources sont réparties en trois catégories. La première est le code décrivant le fonctionnement du logiciel, écrit dans un ou plusieurs langages de programmation. La deuxième catégorie regroupe toutes les ressources nécessaires au fonctionnement du logiciel, par exemple des images, un ensemble de textes traduits dans différentes langues. La troisième catégorie est la description des dépendances d'un logiciel. Un logiciel dépend de bibliothèques, et cette dépendance se retrouve au niveau des sources. La figure 1.1 schématise les concepts décrits. Un logiciel peut avoir plusieurs implémentations au cours de sa vie. Par exemple, `Firefox` cité précédemment a eu plusieurs moteurs de rendu HTML au cours de sa vie. Le logiciel est resté le même dans cette modélisation, avec des sources associées qui changent. Les évolutions du logiciel sont donc associées aux évolutions des sources.

Les sources d'un logiciel sont compilées pour produire du code exécutable par un processeur. Ce code exécutable porte plusieurs noms dans cette thèse :

- exécutable : ce terme correspond au code exécutable associé à un logiciel utilisateur
- bibliothèque exécutable : ce terme correspond au code exécutable associé à une bibliothèque
- binaire : ce terme correspond à l'ensemble du code exécutable, indépendamment de la catégorie du logiciel, il regroupe les deux termes précédents. Il est le fichier qui contient ce code exécutable.
- programme : ce terme correspond à l'ensemble du code binaire exécutable chargé en mémoire d'un logiciel. Il englobe donc l'exécutable et les bibliothèques exécutables dont l'exécutable dépend.

Le processus de transformation des sources d'un logiciel au binaire est appelé compilation. La compilation est modélisée de manière très abstraite dans cette thèse. Un outil utilisé pour réaliser cette transformation est appelé compilateur.

La figure 1.2 présente le procédé de génération de binaire à partir d'un code C simple. Ce code est une unique fonction qui prend une chaine de caractères et un caractère, et renvoie l'index de la première occurrence du caractère s'il est présent, et -1 sinon. Le procédé montre la compilation par les deux compilateurs les plus courants, `gcc` et `clang`. Les différents compilateurs disponibles utilisent un langage interne de description. La compilation est décomposée en deux étapes dans cette thèse. La première est la traduction, représentée en magenta au numéro (1). Cette traduction transpose les sources dans le langage de description interne des sources du logiciel. La représentation dans ce langage de description du logiciel est appelée représentation interne du logiciel. Les compilateurs travaillent sur cette représentation pour effectuer des opérations d'optimisation, ou de sécurisation par exemple. Les représentations internes utilisées dans la figure montrent le résultat après certaines de ces modifications. La seconde étape de la compilation est la génération du binaire, en vert au numéro (2), à partir de la représentation interne du logiciel. Le binaire n'étant qu'une suite d'octets, il est représenté dans un assembleur. Cette représentation permet une lecture aisée du code machine, et est utilisée dans cette thèse comme représentation du binaire. Le résultat du processus de compilation donne un résultat qui diffère d'un compilateur à l'autre, même sur une fonction aussi simple. Cela a des conséquences sur le détournement de flot d'exécution, et cela sera plus développé au chapitre 3 principalement.

Le processus de compilation est paramétrable. En effet, les sources d'un logiciel ne sont pas spécifiques à un compilateur, dans le cas général. Il est possible, parfois, d'utiliser deux compilateurs différents à partir des mêmes sources pour générer deux binaires différents. La compilation dépend de plusieurs éléments supplémentaires, comme le choix d'algorithme d'optimisation et de protection du binaire par exemple. Une autre possibilité porte sur certaines bibliothèques qui sont interchangeables. La sélection d'une des alternatives a une influence sur le binaire produit. Un programme dont les sources sont en C peut dépendre pour la partie standard soit de la bibliothèque C standard GNU soit de la bibliothèque `musl`. Bien que ces deux bibliothèques fournissent les mêmes services, elles ont des particularités qui influent sur le contenu du binaire.

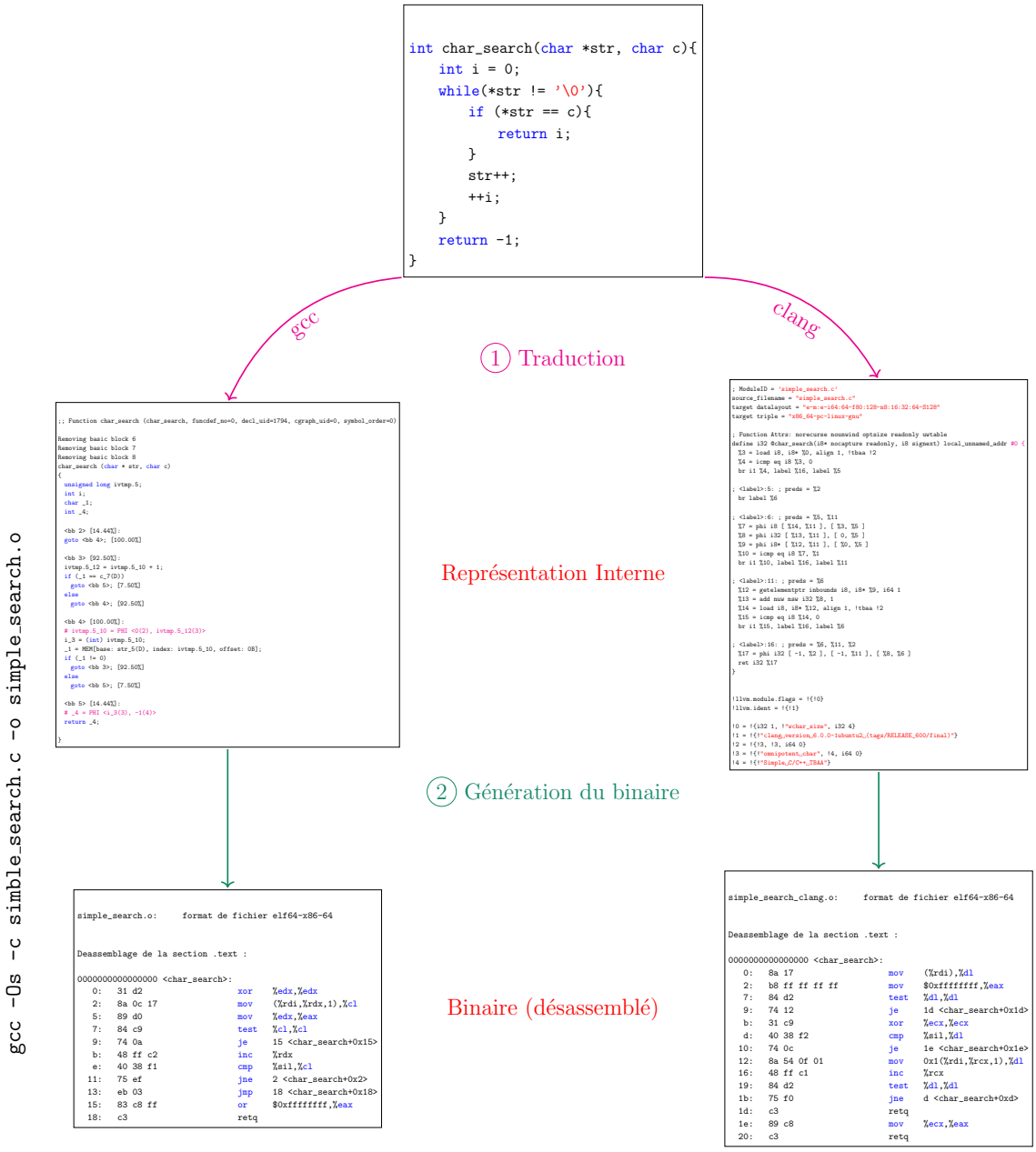


FIGURE 1.2 – Processus de compilation d'un fichier contenant une fonction simple, à l'aide des deux compilateurs `gcc` et `clang`

1.2.1 Cible matérielle

Le binaire contient du code exécutable sur un processeur. Chaque famille de processeurs fournit un ensemble d'instructions valides que chaque processeur de la famille peut exécuter. L'ensemble des instructions qu'un processeur peut exécuter est nommé ISA, pour *Instruction Set Architecture*, dans cette thèse. L'ISA contient aussi d'autres particularités, comme le mode d'adressage. Une famille de processeur va par exemple être l'ensemble des processeurs ARMv7-M. Pour la famille des processeurs x86_64 on peut avoir par exemple le processeur Intel Xeon 3480 ou un AMD Opteron X3000. Les deux processeurs implémentent la même ISA de manière bien distincte, bien que dans un cadre général du code généré pour l'un peut fonctionner sur l'autre. Pour les besoins de cette thèse, peu de détails sont utilisés sur les différentes ISA qui existent. Les ISAs présentes sont classées en deux catégories : RISC et CISC, respectivement *Reduced Instruction Set Computer* et *Complex Instruction Set Computer*. D'autres catégories existent, mais ne sont pas incluses dans cette thèse, car elles ne sont que très peu déployées.

Outre la classification des ISAs par leur fonctionnement et leur richesse en nombre d'instructions, la taille des adresses manipulées sert aussi de classification. Dans cette thèse, les ISAs sont classées dans deux autres catégories : 32 bits et 64 bits. D'autres architectures, plus anciennes notamment, travaillent sur 16 bits ou 24 bits, mais ne sont pas étudiées dans ce document.

1.2.2 Construction d'un système

Un système est l'ensemble d'un ou plusieurs exécutables, bibliothèques, un processeur et l'environnement dans lequel cet ensemble d'exécutables évolue. L'environnement définit l'ensemble des paramètres caractérisant le système. Par exemple, la configuration du noyau et celle des bibliothèques sont prises en compte dans cette caractérisation. Chaque élément du système est nommé composant du système. Un des composants du système est le système d'exploitation. Il gère les interactions entre tous les composants.

La figure 1.3 présente un résumé des composants d'un système. Les systèmes qui sont considérés dans cette thèse peuvent avoir plus d'un millier de programmes. Le chapitre 2 en page 43 présente les différents systèmes considérés.

Un système cible est le système sur lequel le logiciel étudié est déployé. Le compilateur doit produire un binaire compatible avec le système cible. Lorsque le

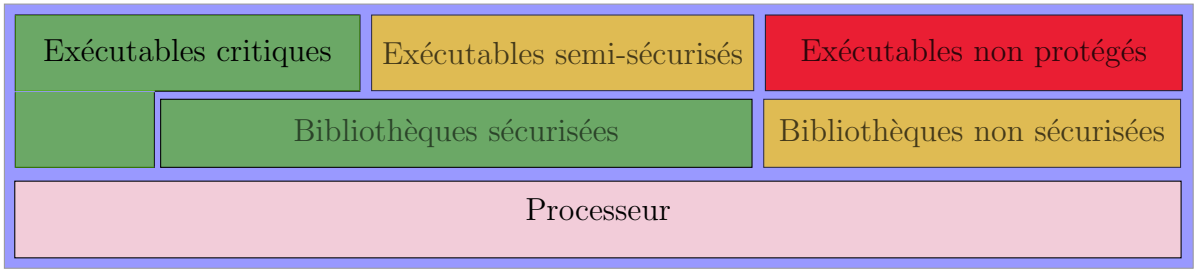


FIGURE 1.3 – Illustrations des différentes configurations de sécurisations sur un système

système sur lequel le binaire est produit n'est pas le système cible, le système de construction est appelé système hôte.

Dans cette thèse, l'objet d'étude est la sécurisation du système cible. Un système sécurisé peut l'être de trois façons. La figure 1.3 illustre les trois catégories, décrites comme suit :

— Protection intégrale du système. Toutes les bibliothèques sont protégées, mais cela à l'inconvénient de détériorer les performances de l'ensemble du système. Toutes les parties critiques des binaires sont ainsi protégées, et donc très peu dupliquées.

— Aucune dépendance pour les logiciels critiques : l'intégralité du code nécessaire à un logiciel critique est intégrée au binaire. On évite ainsi d'avoir à protéger une bibliothèque dont beaucoup de logiciels dépendent, telle que la `libc` par exemple. Cela permet aussi de protéger uniquement ce qui est nécessaire et évite les pertes de performance induites sur des composants non critiques qui dépendent aussi de cette bibliothèque. L'inconvénient principal est une utilisation plus importante de l'espace de stockage, car les binaires critiques peuvent contenir du code exécutable commun qui est donc dupliqué. Cela peut être limitant sur certains systèmes, notamment embarqués, pour lesquels l'espace de stockage un des facteurs les plus contraignants.

— Un système hybride. Le système est construit en deux parties, l'une protégée directement et l'autre non protégée. Les composants critiques sont identifiés et intégralement protégés. La sécurité du système complet repose sur ces composants et les logiciels non critiques présents ne sont pas protégés. Ce type de système est assez courant, avec des noyaux bien protégés et des logiciels utilisateurs non protégés reposant sur ces noyaux.

1.3 Exécution sur une architecture x86_64

Pour comprendre comment se comporte un logiciel en pratique, cette section présente le fonctionnement simplifié de l'exécution de code sur un processeur x86_64. Cet ISA regroupe des processeurs très courants, notamment sur la majorité des ordinateurs de bureau. Bien que cette thèse n'est pas limitée à ces ISAs, cette architecture est adaptée à l'explication des principes mis en jeu dans ce document. Les détails exacts seront laissés de côté, c'est l'exécution haut niveau qui est expliqué ici.

1.3.1 Initialisation de l'exécution

Avant de commencer l'exécution du programme, le système effectue plusieurs actions. La première action précédant l'exécution est l'allocation de la mémoire au programme. La mémoire allouée correspond aux besoins du programme, donc de l'exécutable ainsi que toutes les bibliothèques dont il dépend. L'exécutable est séparé en plusieurs sections. Sur les systèmes GNU/Linux, le format de fichier des binaires est le format ELF (*Executable and Linkable Format*). Un fichier ELF est découpé en plusieurs sections. Les sections principales sont les suivantes :

- `.text` : la partie exécutable du binaire
- `.data` : les données nécessaires à l'exécutable
- `.rodata` : les données de l'exécutable qui doivent être chargées en lecture seule (pas de possibilité de modification).

Lorsqu'une zone mémoire est déclarée par le système d'exploitation, des *flags* y sont associés. Les *flags* dont il est question dans cette thèse sont ceux qui définissent les droits associés à une zone mémoire : lecture, écriture et exécution. La section 1.3.2 présente plus en détail cette notion.

La section `.text` contient donc tout le code qui peut être exécuté par le processeur. Lorsque le système charge le programme, la plage mémoire est marquée sans écriture et l'exécution y est autorisée. La section `.data` ne contenant que des données, qui peuvent être modifiées au cours de l'exécution, la zone est donc marquée comme ouverte en écriture et non exécutable. Enfin la section `.rodata` qui contient tous les données non modifiables, souvent les chaînes de caractères du programme. Les données sont chargées dans une plage que le système marque comme non modifiable et non exécutable.

D'autres sections sont présentes dans le binaire, mais présentent moins d'intérêt.

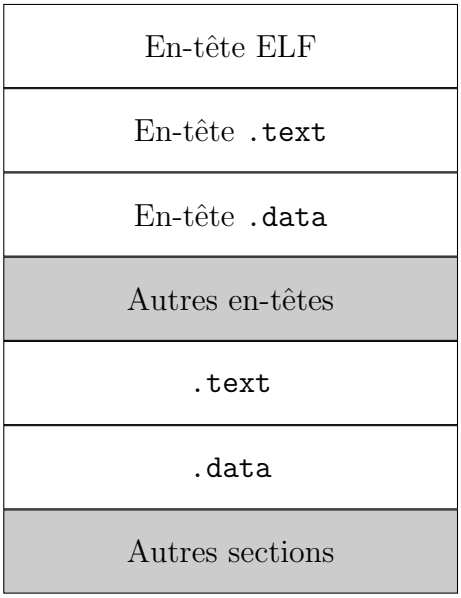


FIGURE 1.4 – Exemple de structure d'un fichier ELF

Par exemple la section `.comment` peut contenir des informations inscrites par le compilateur. Ces informations sont généralement chargées en lecture seule, mais parfois elles ne sont pas chargées du tout. Ensuite une section souvent présente est la section `.bss`. Cette section est utilisée pour les variables et tableaux non initialisés du programme, le contenu est donc à 0. Lorsqu'un système charge le programme, normalement, le contenu de cette section n'est pas copié, et la plage mémoire est automatiquement mise à 0. Enfin une section courante dans certains environnements est la section `.stab` qui contient les symboles de débogage.

En plus des sections présentées ci-dessus, un certain nombre d'en-têtes descriptifs des sections sont présents dans le fichier. La figure 1.4 donne un exemple de structure de fichier ELF dans un cas général simplifié.

Bien que cette structure ne soit valable que sur les systèmes GNU/Linux, les autres systèmes d'exploitation ont chacun un système de fichiers exécutables normé dans lequel on retrouve les mêmes notions de section. Par exemple, le format utilisé sur les systèmes Windows modernes est le format PE (*Portable Executable*) et les systèmes Mac OS utilisent le format Mach-O.

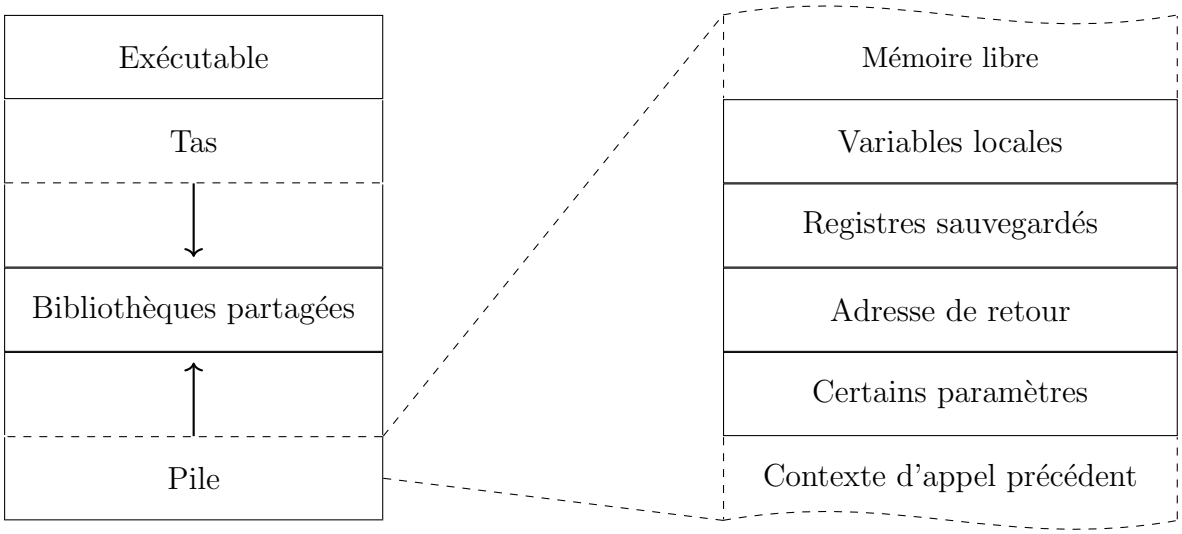


FIGURE 1.5 – Cartographie de mémoire d'un programme

1.3.2 Cartographie de la mémoire

La mémoire qu'un système alloue à un programme est structurée par la fonction de chaque zone. Ces zones sont bien délimitées pour permettre la mise en place de *flags* de protection matérielle propres à chaque zone. La cartographie mémoire d'un programme est représentée en figure 1.5. Cette figure montre où se trouve chaque zone mémoire d'un programme, en ne présentant que les zones principales. Les deux flèches représentent le sens dans lequel le système alloue de la mémoire au tas et à la pile. Cette représentation simplifiée est vérifiée sur les systèmes GNU/Linux x86, bien que d'autres systèmes peuvent avoir une cartographie similaire. La zone marquée comme `Exécutable` dans cette figure représente l'ensemble des sections `.text`, `.data`, `.bss` et `.rodata` du binaire. Chaque bibliothèque chargée par le système va aussi avoir des zones mémoires avec différents *flags* en fonction de l'utilisation de la zone.

Le tas est la zone mémoire dans laquelle l'allocation dynamique de mémoire est réalisée. La plupart des données manipulées par le programme y sont présentes. La pile est la zone mémoire dans laquelle se trouvent les données utilisées pour contrôler l'exécution du programme. L'allocation de mémoire est très structurée. La figure 1.5 présente aussi l'agencement général de la pile. Le rôle de chaque élément est précisé dans la section 1.3.3. L'allocation de mémoire sur la pile est faite lors

d'un appel de fonction, et la libération de cette mémoire au retour de cet appel de fonction. Il n'est généralement pas à la charge du programme de gérer cette allocation.

1.3.3 Déroulement d'un appel de fonction

Un processeur est composé de plusieurs registres permettant d'exécuter un programme. L'ISA du processeur définit des instructions lui permettant d'agir sur les différents registres. Ces actions permettent, entre autres, d'effectuer des opérations arithmétiques, de déplacer des données ou diriger l'exécution d'un programme. Afin d'illustrer la correspondance entre le fonctionnement du processeur et ce qui est écrit au niveau des sources, la figure 1.6 présente le déroulement d'un appel de fonction en C et sa traduction pour un processeur de la famille x86_64. Cet exemple est utile pour comprendre le fonctionnement des attaques décrites dans cette thèse. Il est notamment repris en section 1.4.

L'exécution se déroule en plusieurs étapes. La première est la préparation de l'appel de la fonction. L'étape principale d'un appel de fonction, du point de vue du processeur, est la sauvegarde de l'adresse mémoire de retour de la fonction. Cette adresse est sauvegardée sur la pile d'exécution. Cette étape est illustrée par la première flèche sur la figure. La mémoire pour la sauvegarde de l'adresse est allouée (l'adresse maximale diminue de 8) et l'adresse de retour y est inscrite (`main+42`). Ensuite, sur la pile, au-dessus de cette adresse, le processeur enregistre les autres paramètres, et met à jour les registres contenant les adresses des différentes piles (précédente et courante). La figure illustre ces opérations avec les flèches numérotées 2 à 4. Lors de la sortie de la fonction, la valeur de retour est enregistrée ici dans le registre `%eax`, mais cela ne change pas le statut de la pile. Le processeur restaure ensuite la pile dans l'état précédent l'appel de la fonction, illustré par les flèches 6 et 7. La dernière instruction réalisée ici est `ret`, qui récupère l'adresse de retour sur le dessus de la pile pour la placer dans le registre `%rip`, qui indique l'adresse de la prochaine instruction à exécuter.

L'exécution du programme par le processeur est séquentielle entre deux instructions qui permettent de modifier ce registre `%rip` sur x86_64. Chaque instruction a une taille donnée, et le registre `%rip` est incrémenté de cette taille. Certaines instructions, comme `call` et `ret` montrées ci-dessus, modifient ce registre en y inscrivant directement une valeur. L'exécution est donc modélisée sous forme d'un graphe, nommé graphe d'exécution, graphe de contrôle nommé en anglais *control-flow*

```
;int main () {
;    char arr[16];

;    funct(arr);
  36:   48 8d 45 f0     lea    -0x10(%rbp),%rax
  3a:   48 89 c7        mov    %rax,%rdi
  3d:   e8 00 00 00 00  callq  0x400546 <funct>
;    printf("%s", arr);
  42:   48 8d 45 f0     lea    -0x10(%rbp),%rax
; ...
```

```
;int funct (char * stack_arr) {
   0:   55               push   %rbp
   1:   48 89 e5         mov    %rsp,%rbp
   4:   48 83 ec 10      sub    $0x10,%rsp
; ...
;    return 0;
  27:   b8 00 00 00 00   mov    $0x0,%eax
;}
  2c:   c9               leaveq
  2d:   c3               retq
```

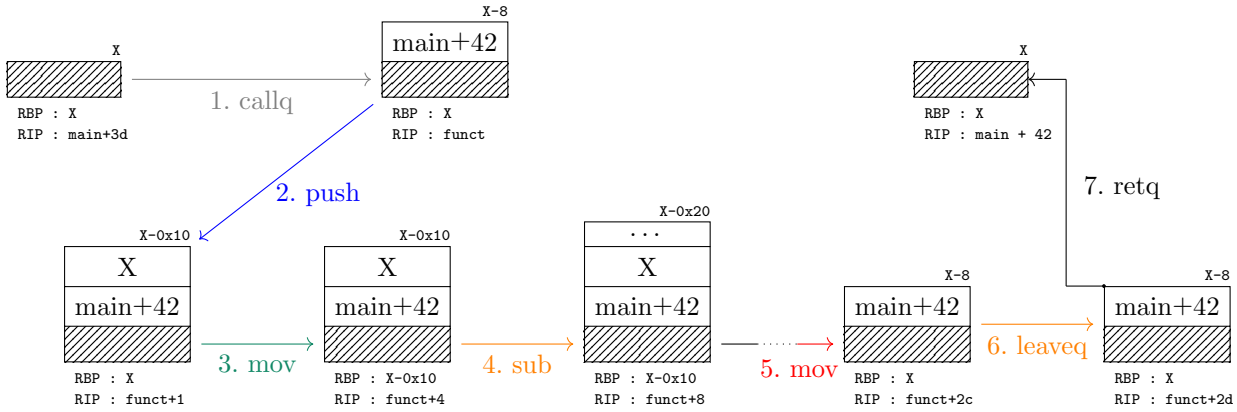


FIGURE 1.6 – Exécution d'un appel de fonction sur x86_64, effet sur la pile et certains registres

graph, où chaque branche représente l'utilisation d'une instruction de contrôle de flot d'exécution et chaque nœud représente le code séquentiel entre deux de ces instructions, nommé bloc basique.

Un logiciel a un graphe d'exécution à plusieurs niveaux. Dans un premier temps ce graphe peut être décrit au niveau des sources. Un appel de fonction en C par exemple va définir un saut d'un bloc à un autre. Lors de la compilation, d'autres chemins peuvent être créés ou supprimés. Certains paramètres de la compilation suppriment des chemins pour améliorer les performances du logiciel. Dans cette thèse, le graphe de contrôle étudié est celui du binaire, cela est expliqué au chapitre 2 en page 43. Les appels de fonctions sont traduits en branchements.

1.4 Détournement de flot d'exécution

Le concept de détournement de flot d'exécution d'un logiciel représente l'ajout dans le graphe de contrôle de chemins non présents à la fin de la génération d'un binaire. Ces détournements reposent sur plusieurs éléments. Le premier élément nécessaire est l'utilisation d'une corruption mémoire. La section 1.5 présente des moyens d'y arriver. Ces corruptions mémoires peuvent avoir différents effets. Les effets décrits ici le sont pour des corruptions qui sont provoquées volontairement par l'attaquant. Une protection présente sur tout système sécurisé est la prévention d'exécution de données, DEP pour *Data Execution Prevention*. Cette protection permet d'empêcher l'exécution de code par le processeur dans des zones mémoires ouvertes en écriture et donc sensibles aux corruptions mémoires. L'injection de code arbitraire en mémoire par un attaquant n'est donc pas possible dans le modèle d'exécution considéré. Les fichiers binaires d'un logiciel contiennent les informations permettant de définir les zones mémoires marquées comme exécutables et celles marquées comme ouvertes en écriture. Une zone étant exclusivement l'un ou l'autre, cette protection est nommée W⊕X pour *Write xor Execute*. Pour étudier le flot d'exécution d'un programme dans un modèle permettant les corruptions de mémoire, la notion de code légitime est définie comme suit :

Définition. *Le code légitime est du code alloué au programme, dans une zone mémoire accessible et exécutable (conforme à l'ISA) et avec le bon* flag *W⊕X.*

Le détournement de flot d'exécution désigne l'exécution de code légitime d'un programme d'une manière non prévue à la conception du logiciel. Lors d'un détournement de flot de contrôle, un attaquant ajoute dans le graphe de contrôle du binaire un chemin d'exécution en utilisant des corruptions mémoires. Une méthode courante de détournement de flot de contrôle est le *Return Oriented Programming* ou ROP. Cette méthode a été décrite par Shacham [Sha07]. Le ROP est un paradigme qui permet la génération de nouveaux logiciels en n'utilisant que du code exécutable du binaire attaqué. La première description concerne l'architecture x86, mais a été généralisée par la même équipe sur des architectures RISC, notamment SPARC [Buc+08]. Le principe du paradigme est d'utiliser des suites d'instructions qui précèdent des instructions `ret` pour pointer vers d'autres suites dans un enchaînement non prévu. Ce détournement est représenté en figure 1.7. Cette figure présente des blocs basiques en gris avec un flot d'exécution en bleu clair, prévu par le développeur. Un attaquant qui utilise le paradigme ROP utilise les blocs en bleu hachurés. Le flot d'exécution possible est représenté par les flèches

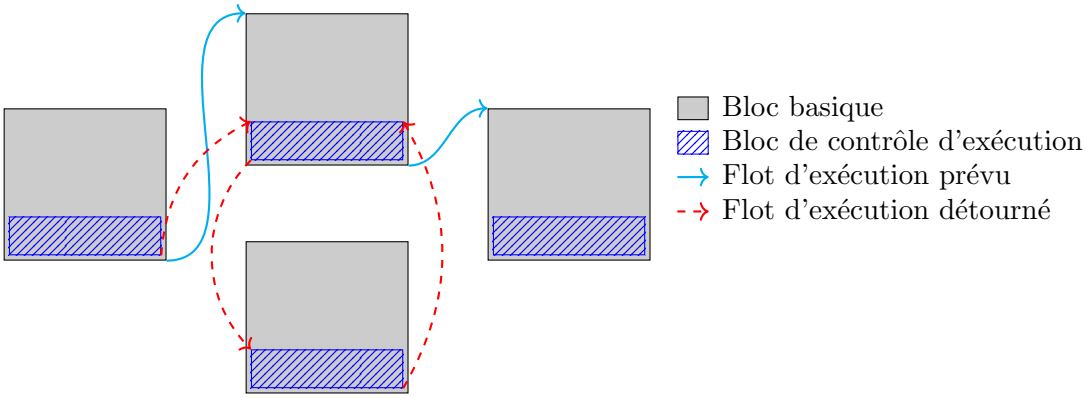


FIGURE 1.7 – Exemple de flot d'exécution prévu et détourné dans un programme

en pointillés rouges.

1.4.1 Définition d'un gadget

Un gadget est défini de différentes manières dans la littérature. La forme d'un gadget peut différer légèrement d'un article à l'autre, en fonction des éléments recherchés par les auteurs. Un gadget est défini par la suite en se basant sur les définitions existantes.

La première définition d'un gadget est décrite chez [Sha07]. Cette définition est généralisée ensuite pour inclure d'autres architectures matérielles [Roe+12]. Les gadgets y sont décrits comme l'ensemble des instructions disponibles à un attaquant pour construire son attaque. Le gadget est défini comme une suite d'instructions terminée par un saut. La définition de l'ensemble des gadgets comme le catalogue des instructions disponibles pour construire une attaque n'est pas reprise ensuite, bien que souvent implicite. La définition d'un gadget comme suite d'instructions exécutables terminée par une instruction permettant de diriger le flot de contrôle est souvent reprise [BP19 ; PR12 ; Lu+11 ; Bit+14 ; His+12 ; Mur+14]. Enfin, certains se contentent d'une définition peu précise [CBG17 ; DKW10 ; FBB16 ; Bur+17 ; Jac+14 ; Koc+18], par exemple « extraits de code utiles ».

Il est assez régulier dans la littérature de trouver des articles qui ne définissent pas clairement ce qu'est un gadget, considérant cette notion comme implicite. Cependant, le traitement fait des gadgets permet de déduire une définition de gadget comme suite d'instructions exécutables terminée par une instruction de

contrôle de flot.

La définition de gadget qui est utilisée dans cette thèse est regroupée comme suivant :

Définition. *Un gadget est une séquence d'instructions légitimes d'un programme terminée par une instruction, dite pivot, permettant de contrôler le flot d'exécution du programme. La partie du gadget précédant l'instruction pivot est appelée corps du gadget.*

Pour le gadget suivant `lea 0x4(%rbx), %rax; pop %rbx; pop %rbp; ret`, le corps du gadget est `lea 0x4(%rbx), %rax; pop %rbx; pop %rbp;` et l'instruction pivot est `ret`.

Dans la littérature, plusieurs termes différents sont utilisés pour nommer l'instruction pivot. Un terme utilisé régulièrement est le terme d'instruction de saut. Ce terme est moins général, car toutes les instructions pivots ne sont pas toutes des instructions de saut, notamment l'instruction `syscall`.

Lors de l'utilisation des gadgets pour construire une attaque, l'intégralité des instructions qui le composent n' est pas forcément utile à un attaquant. En effet, l'attaquant peut ne pas avoir de choix sur les gadgets dont il dispose, notamment dans les binaires de taille inférieure à 10 ko. Dans cette thèse, et de manière cohérente avec ce qui est fait dans la littérature, les effets parasites introduits par les instructions non voulues du gadget sont nommés effets de bord. Les effets considérés dans ces effets de bords sont la modification de registre et la modification de zone mémoires non utiles à l'attaque. Cela peut causer des problèmes en laissant des traces ou rendre plus difficile la suite de l'exploitation avec d'autres gadgets.

1.4.2 Catégorisation des gadgets

Un gadget peut permettre de faire différents calculs, comme par exemple des calculs arithmétiques, des accès mémoire, etc. Pour les architectures de la famille x86, quatorze catégories sont sélectionnées en accord avec la référence disponible ici [1]. Ces catégories peuvent être adaptées à d'autres architectures, même si elles n'ont pas toutes de correspondance. Les catégories d'instructions sont les suivants :

— Arith, exemple : `add 0xb5,%rax`
  Ajout de la valeur `0xb5` au registre `%rax`

1. http://ref.x86asm.net/ (visité le : 2020-08-20)

— Data Move, exemple : `lea -0x10(%rbp),%rax`
Chargement d'une valeur en mémoire située à l'adresse `%rbp-0x10` dans le registre `%rax`
— Logic, exemple : `xorl %edx, %ecx`
Ou Exclusif entre les deux registres, le résultat est dans le registre `%ecx`
— Shift and Rotate, exemple : `shldw $8,%bx,%ax`
Déplacement vers la gauche de 8 bits du registre `%bx`, inscrit dans le registre `%ax`
— Flag, exemple : `pushf`
Sauvegarde le registre `EAFLAG` sur le dessus de la pile
— NOP, exemple : `nop`
Pas d'opération sur ce cycle
— Floating Point, exemple : `fmul %st0, %st0`
Multiplication flottante du registre flottant `%st0` par lui-même
— Misc, exemple : `CPUID`
Récuparation d'information du procésseur.
— I/O, exemple : `outsw`
Écrit une chaîne sur un port. Ici le registre d'entrée est `%rsi` et le port de sortie est `DX*`
— Segreg, exemple : `lds %r9d, ptr [%rax - 0x77]`
Charge un pointeur lointain (spécifique i386). Registre de segment utilisé `DS`
— Branch, exemple : `je 0x423d24`
Saute à l'adresse indiquée si le bon flanion est à 1 dans le registre `EAFLAG` (ici le flag Z : zero flag)
— System, exemple : `sysexit`
Sort d'un appel système
— Conver, exemple : `CQO`
Converti une valeur de 16 bits en 32 bits
— Break, exemple : `INT1`
Génère un piège de débogage

D'autres catégories d'instructions pourraient être considérées, mais ne sont pas pertinentes dans une optique de sécurité. En effet, les instructions MMX ou SIMD ne sont pas considérées l'instant, dans un contexte d'attaque par détournement de flot d'exécution. Cela correspond à ce qui est fait dans la littérature, comme par Follner et. al. [FBB16]. L'ajout de ces instructions pourraient être considéré si les attaques évoluent vers l'utilisation de ce type d'instructions.

Un gadget est caractérisé par l'ensemble des catégories de chacune des instructions du corps du gadget, sans ordre particulier. La catégorisation de l'instruction pivot est séparée de la catégorie du corps. Ansi un gadget peut être catégorisée arith, et l'instruction pivot permet de le classer par la suite du côté ROP pour les pivots `ret`, COP (*Call Oriented Programming*) pour les pivots `call`, JOP (*Jump Oriented Programming*) pour les pivots `jmp` et les autres pivots comme `INT` ou `syscall`. Les instructions pivots sont présentes dans les catégories `break`, `branch` et `system`.

Voici deux exemples de catégorisation de gadgets :

— `lea 0x4(%rbx), %rax; pop %rbx; pop %rbp; ret`
Gadget ROP ayant une seule catégorie : `[Data Move]`

— `lds r8, ptr [rax] ; xor eax, eax ; call 0x58857`
Gadget COP de catégories `[Segreg, Logic]`

1.4.3 Gadgets non-alignés

Lors de la recherche de gadgets, sur certaines architectures, du code légitime peut être présent sans correspondre à du code source écrit par le développeur. En effet, tout code situé dans une zone mémoire exécutable est considéré comme légitime du point de vue du processeur. La seule caractéristique qui est vérifiée est la présence du *flag* autorisant l'exécution. L'architecture i386 et sa variante en 64bits x86_64, permettent au processeur de faire des lectures arbitraire en mémoire, sans restriction sur l'alignement des adresses. C'est une fonctionnalité rarement présente sur d'autres architectures qui ne permettent que des lectures alignées sur un certains nombre de bits, 32 bits pour SPARC par exemple.

La possibilité d'utiliser du code non aligné sur x86_64 permet à un attaquant d'utiliser bien plus de gadgets que ce qui existe dans le code source du logiciel. Prenons le code suivant, représenté en figure 1.8 :

```
lea 0x4(%rbx), %rax; pop %rbx; pop %rbp; ret
```

Ce code en mémoire a la forme suivante, en hexadécimal, `488d43043045b5dc3`. Avec une lecture décalée de trois octets, on obtient donc `045b5dc3`, qui est interprété par le processeur comme étant la séquence :

```
add 0x5b, %al; pop %rbp; ret
```

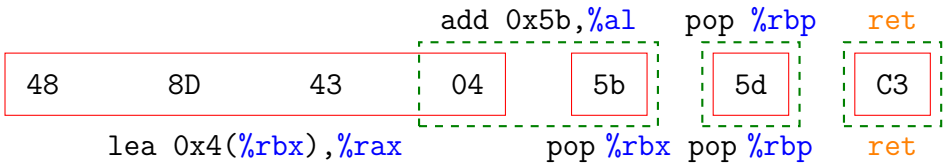


FIGURE 1.8 – Exemple de lecture décalée, sur x86_64, pour trouver de nouveaux gadgets

Le code prévu pour le programme est en rouge, le gadget obtenu en lisant le code avec un décalage de trois octets est encadré en vert pointillé. Ce nouveau gadget est valide du point de vue du processeur, toute la zone dans lequel il se trouve est exécutable, non ouverte en écriture. Le code est légitime. Dans cet exemple, le gadget utilise la même instruction pivot pour diriger le flot d'exécution de l'attaque, mais ce n'est pas toujours le cas. En effet, certaines instructions peuvent contenir la valeur de l'instruction `ret`. C'est le cas par exemple pour l'instruction `add %eax,0xC3AA`, réutilisée pour la description de GFree en section 1.7.3. L'adresse utilisée dans cette instruction contient l'octet 0xC3 qui correspond à l'instruction `ret`.

La notion de gadget non-aligné est nécessaire du point de vue conception du logiciel. Pour l'attaquant, lorsqu'il examine le binaire pour y trouver des gadgets, il n'est pas possible de distinguer les gadgets alignés et non-alignés. La recherche de gadget se fait en cherchant une instruction qui peut servir de pivot et regarder si les octets précdents cette instructions sont interprétables et légitimes. La propriété d'alignement du gadget n'est pas intrinsèque à celui-ci.

L'architecture SPARC par exemple ne permet pas ce type de lecture en mémoire, toute adresse de code devant être alignée sur 32 bits. L'architecture ARM, en 32 bits, propose de passer le processeur en mode 16 bits. Toute la zone exécutable peut donc être lue soit en 32 bits soit en 16 bits. Bien que ce ne soit pas aussi permissif qu'avec les architectures i386 et x86_64, cela permet éventuellement de trouver des gadgets supplémentaires.

1.4.4 Construction d'une attaque par détournement de flot d'exécution

Cette section présente comment une attaque est construite à partir des gadgets présentés jusqu'ici. Un exemple de charge d'attaque et des éléments la composant sont donnés pour illustrer ce que va chercher un attaquant. La charge d'une attaque

par détournement de flot d'exécution est appelée chaîne de détournement de flot d'exécution et est définie comme suit :

Définition. *Une chaîne de détournement de flot d'exécution est la suite des adresses des gadgets et des données supplémentaires réalisant l'exploitation du programme par un attaquant. Les données supplémentaires sont par exemple des valeurs nécessaires pour les calculs ou des chaînes de caractères.*

Prenons un exemple. Un attaquant cherche à simplifier une attaque, et pour cela il décide de désactiver les protections en place. Cet exemple montre une chaîne qui permet de désactiver le DEP sur la zone mémoire de la pile d'exécution. La chaîne est la suivante, où chaque `.quad` représente une valeure de 64 bits (*quad word*) :

```
.quad _stack_pointer & ~ 0xfff
.quad 0x7e43b
.quad 7
.quad 0x1000
.quad 0x4eb8c
.quad 0
.quad 0
.quad 0
.quad code
```

Dans cette chaîne 3 adresses sont présentes

- `0x7e43b` : adresse vers le gadget `pop %rdx; pop %rsi; ret`
- `0x4eb8c` : adresse vers un gadget d'appel de la fonction `mprotect`
- code : adresse vers le code qui est injecté pour réaliser le corps de l'attaque (pour lequel la protection DEP est désactivée)

Cette chaîne a besoin de ces deux gadgets. Le premier gadget permet de mettre les valeurs nécessaires en paramètres d'un appel système. L'attaque effectue l'appel `mprotect(stack, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC)`, suivi de l'exécution d'un code compilé statiquement écrit sur la pile à la suite de la chaîne de détournement de flot d'exécution. Pour cela les deux registres visés sont `%rdx` et `%rsi`. L'attaque ayant été initiée par un débordement de tampon sur la pile (expliqué en section 1.5), les deux valeurs sont elles aussi présentes sur la pile : 7 pour le bon paramètre d'appel système et `0x1000` pour la taille de la zone rendue exécutable. Cette chaîne est similaire dans son fonctionnement au code assembleur suivant :

```
movl 7, %edx
movl 4096, %esi
movabsq 0x7ffffffe000, %rdi
call mprotect
```

Cet exemple comporte des adresses statiques (0x7e43b et 0x4eb8c) qui ont été cherchées dans le binaire connu de l'attaquant. Pour que cette chaîne soit utilisable en l'état, une protection a été désactivée. Cette protection est l'*Address Space Layout Randomization (ASLR)*, et est décrite en section 1.7.1.

Pour construire une attaque, deux points sont à considérer. Premièrement, il faut connaître exactement quelles sont les instructions nécessaires à l'attaque. Par exemple, si le but de l'attaque est de désactiver le DEP comme dans notre exemple, l'attaquant a besoin de créer un appel système avec les bons paramètres. Ensuite, il faut vérifier que les gadgets présents dans le programme attaqué contiennent les instructions de l'attaque. Si aucun gadget ne contient les instructions nécessaires, il va falloir reconstruire différemment les gadgets manquants. Pour, cela d'autres gadgets peuvent être considérés pour arriver au même résultat avec plus d'étapes intermédiaires. Cela peut cependant avoir des effets de bord. L'attaque effectivement réalisée peut légèrement différer de ce qui devait être fait initialement, du fait de l'utilisation de ces gadgets à effet de bord. Un attaquant peut choisir ses gadgets en fonction de ce qui l'intéresse pour son attaque, et les effets de bord qu'il veut éviter. Ainsi, un attaquant a la possibilité de construire plusieurs chaînes de détournement de flot d'exécution réalisant le même objectif.

Prenons un exemple. Une certaine valeur est nécessaire dans le registre `%rdx` et un gadget de longueur 1 permettant cette écriture, comme `pop %rdx; ret` n'est pas disponible. Si ce gadget n'est pas disponible, d'autres gadgets doivent être utilisés pour arriver au même résultat. Le gadget `pop %rdx; add %rbx,%rdx; ret` est un candidat possible. D'un point de vue fonctionnel pour l'attaque, ce gadget ne change pas le résultat voulu. Par contre, il a une influence sur un autre registre que celui qui intéresse, `%rbx`.

1.5 Corruptions mémoires

La gestion de la mémoire est critique pour la sécurité des programmes. Pour qu'un attaquant puisse détourner le flot d'exécution d'un programme, une corruption de la mémoire est nécessaire. La corruption mémoire est définie comme la

modification imprévue d'une zone de mémoire allouée au programme. Ces modifications sont dues généralement à des erreurs de programmation dans le cadre de corruptions logicielles, comme le présente la section 1.5.1. Éviter les erreurs de programmation ne suffit pas car il est possible de provoquer des corruptions, à la fois au niveau logiciel [GMM16] ou du matériel [Kim+14 ; HTI97]. Cette section présente des exemples d'exploitation de corruptions mémoires.

1.5.1 Débordement de tampon sur la pile

Le premier exemple de corruption est le débordement de tampon sur la pile d'exécution. Cette corruption est utilisée pour réécrire la pile d'exécution notamment pour y modifier les adresses de retour de fonction qui y sont présentes, comme cela est montré en section 1.3.2. Pour qu'une telle corruption soit possible, une fonction d'écriture ne doit pas avoir de vérification sur la taille disponible dans le tampon d'arrivée. Par exemple, la fonction de conversion et de transformation de chaîne de caractères `sprintf` (dont l'usage est maintenant déconseillé) peut être problématique dans sa version initiale, qui n'a pas de telle vérification. Une utilisation de cette fonction peut être la saisie d'une information par un utilisateur. Le code C résultant va ressembler à quelque chose comme :

```
char *out;
//...
sprintf(out, "%s", str_in);
```

Si la taille allouée au tampon `out` est plus grande que la taille de la saisie de l'utilisateur, tout va bien. Si l'utilisateur est malveillant, il peut saisir ici une entrée `str_in` plus grande que le tampon de sortie, et l'écriture va déborder à l'extérieur du tampon. Lorsque ce tampon est sur la pile d'exécution, un attaquant va pouvoir écrire sur cette pile jusqu'à écraser l'adresse de retour de la fonction. Le processeur n'a pas de notion de taille de tampon. La sémantique de la mémoire du processeur est trop limitée pour qu'il se rende compte seul des dépassements de tampon. La vérification de la légitimité des écritures réalisées est donc à la charge du développeur.

1.5.2 Corruptions mémoires temporelles

Le débordement de tampon est une erreur spatiale d'écriture, suite à un défaut de protection de pointeur. Les pointeurs peuvent être à l'origine d'autres corruptions. Un pointeur a une certaine validité dans le temps. Par exemple, si un pointeur est alloué sur la pile lors de l'appel d'une fonction, à la sortie de cette fonction, la mémoire est libérée. Cette zone mémoire peut cependant être référencée ailleurs dans le code, sans la connaissance de l'invalidité du pointeur. Le terme utilisé pour ces corruptions est *Use-After-Free*.

Les erreurs d'utilisation temporelles d'un pointeur sont un problème critique en C++, notamment lors de l'utilisation de la `vtable`, un tableau de pointeurs pour fonctions virtuelles. Schuster *et.al.* [Sch+15] montrent l'utilisation de corruptions de ces tables pour mener des attaques sur des logiciels.

1.5.3 Corruption matérielle : rowhammer

Les deux corruptions présentées précédemment sont logicielles, et sont exploitables à cause d'erreurs et maladresses dans les sources du programme. Certains langages sont conçus pour ne pas être sujets à ce type de corruptions mémoires, par exemple `OCaml` et `rust`. Cependant des corruptions d'origine matérielle existent sur lesquelles un développeur n'a aucun contrôle. Cette section présente brièvement les corruptions de la famille *Rowhammer*, bien que ce ne soit pas la seule famille de corruptions matérielles qui existent, comme par exemple l'injection de faute matérielle[HTI97].

Un bit physique est un transistor chargé ayant pour valeur 1 ou non chargé pour la valeur 0. Les mémoires fonctionnent en terme de lignes, un ensemble de valeurs sont adressées en même temps. Lorsque que des valeurs sont écrites sur une ligne, un faible champ électromagnétique est généré autour des bits qui changent de valeur. Lorsque trop de changements rapides sont effectués sur une même ligne sans que la ligne voisine ne soit rafraichie en même temps, le champ généré peut provoquer des corruptions des données sur la ligne voisine. La démonstration du fonctionnement de ces corruptions a été faite par Kim Yoongu *et. al.* [Kim+14]. L'utilisation de ces corruptions dans des attaques réelles a été montrée, comme par exemple Xiao *et. al.* [Xia+16] pour de l'escalade de privilège au travers de l'isolation d'une machine virtuelle.

1.5.4 Corruption et détournement de flot d'exécution

Toutes ces corruptions permettent de créer un point d'entrée d'une attaque par détournement de flot d'exécution. Comme expliqué précédemment, une corruption est nécessaire pour dévier un programme de son exécution en dehors du graphe de contrôle. Les corruptions mémoires sont à l'origine de plus de 70 % des vulnérabilités majeures du projet Chromium [2], comme l'a aussi ennoncé un ingénieur de Microsoft [Mil19]. Il est difficile de se protéger et de prouver l'absence de toute corruption mémoire. Cette thèse ne traite pas du problème de ces corruptions, seules les attaques par détournement de flot d'exécution sont l'objet d'étude.

1.6 Modèle de menace

Cette section présente le modèle d'attaquant considéré dans cette thèse et les moyens qui lui sont laissés. Pour justifier ce modèle, cette section présente aussi dans quel cadre le développement d'un système est considéré.

Un attaquant qui réalise une attaque par détournement de flot d'exécution se base sur plusieurs éléments. Dans un premier temps il a besoin d'un accès en écriture en mémoire. Que ce soit matériel ou logiciel, l'attaquant a un accès important au système attaqué. Ce modèle est justifié par la possibilité d'avoir un opérateur malveillant ou une tentative d'ingénierie inverse. Ce modèle est décrit dans la littérature sous le nom de *Man-At-The-End (MATE)*.

Dans le modèle de menace utilisé dans cette thèse, les capacités de modification de la mémoire qui sont attribuées à l'attaquant sont les suivantes :

- Accès complet en lecture à la mémoire du programme attaqué
- Accès en écriture dans la mémoire du programme limité aux plages d'adresses ayant le *flag* W⊕X correct
- Les plages d'adresses marquées comme exécutables ne sont pas modifiables par un attaquant

Cet ensemble de capacités est cohérent avec ce qui existe dans la littérature, ce qui est présenté dans la section suivante.

2. https://www.chromium.org/Home/chromium-security/memory-safety (visité le : 2020-10-20)

1.7 Protections

Plusieurs approches sont disponibles pour contrer, au moins en partie, ces attaques. Des protections commencent à être déployées, mais toutes ont des inconvénients à prendre en compte. Dans cette section, les protections disponibles sont présentées. Les protections discutées ici concernent l'exploitation d'un logiciel par détournement de flot d'exécution. Le durcissement de l'espace mémoire pour éviter la présence de point d'entrée n'est pas l'objet de cette section.

1.7.1 *Address Space Layout Randomisation (ASLR)*

Les attaques par réutilisation de code sont basées sur la connaissance du binaire attaqué : les gadgets, les adresses de ceux-ci et les symboles présents dans le binaires sont les éléments recherchés par un attaquant. L'ensemble des informations comme la version, les moyens de compilation, les bibliothèques liées sur le système et paramétrages forment l'ensemble de la connaissance permettant ces attaques. Pour cacher les informations sur les adresses des gadgets dans les binaires, agencer l'espace mémoire d'un programme de manière aléatoire est une solution intuitive. C'est la solution proposée par le projet PAX en 2001 [3].

La solution ajoute un aléa aux adresses de différentes plages. Trois nombres aléatoires différents sont ajoutés aux adresses de trois catégories. Toutes les adresses d'une catégorie sont donc décalées du même *offset*. Ces trois catégories sont :

- Les parties exécutables : .text/.bss/.data
- La pile d'exécution
- Le reste :
  - les bibliothèques liées dynamiquement
  - le tas
  - les zones de mémoires partagées
  - les piles des différents *threads*

Cette protection a été déployée sans difficulté car elle présente une perte de performance négligeable. De plus, le déploiement est fait au niveau du système d'exploitation, les programmes n'ont besoin que d'un léger changement de paramétrage lors de la compilation pour permettre l'utilisation de l'ASLR. Cela se fait en passant une option au compilateur.

Plusieurs attaques sont cependant connues contre cette protection [Bit+14 ;

3. https://pax.grsecurity.net/docs/aslr.txt

Sot07]. Les deux attaques mettent en évidence des moyens de faire fuiter des informations sur les *offsets* utilisés. Cette fuite d'information permet de calculer les adresses des gadgets, diminuant ainsi l'efficacité de la solution. Des techniques modernes permettent de limiter la fuite d'information, comme MARDU [Jel+20]. Le faible coût en performance de l'ASLR basique justifie cependant son déploiement car la protection complexifie fortement l'écriture d'attaque, sans les empêcher. Cette protection est déployée par défaut sur tous les systèmes d'exploitation modernes (GNU/Linux, MacOS, Windows, OpenBSD, ...).

1.7.2 Graphe de contrôle de flot d'exécution et intégrité

Une notion souvent utilisée pour contrôler l'exécution d'un programme est le graphe de contrôle de flot d'exécution, nommé dans la littérature anglophone *control flow graph* et régulièrement traduit en graphe de flot de contrôle. Le programme est divisé en blocs d'exécution séquentielle avec un point d'entrée et un point de sortie. Ces blocs sont appelés *basic bloc*. Ces ensembles d'instructions sont déterminés statiquement lors de la compilation du logiciel.

Ces blocs servent à modéliser les exécutions possibles d'un programme. Ce modèle est nommé graphe de contrôle. Un certain nombre de protections sont basées sur la vérification du comportement du programme exécuté vis-à-vis du modèle d'exécution. Ce contrôle peut être fait de différentes manières, soit par le binaire lui-même, soit par un moniteur extérieur. Les deux méthodes ont des avantages et inconvénients qui seront discutés au cas par cas. Cette section présente quelques solutions de CFI (*Control Flow Integrity*), bien que plus d'une trentaine de déclinaisons du principe est disponible dans la littérature, par exemple rien qu'en 2020 [Jun+20 ; He+20 ; JPL20 ; Mau+20]. Le choix des protections décrites dans cette section est motivé pour montrer une diversité et une évolution du concept de CFI.

Control Flow Integrity

La première proposition de vérification d'intégrité du graphe de flot de contrôle est présentée en 2009 par Abadi *et. al* [Aba+09]. Ils introduisent le terme de *Control Flow Integrity*. Dans leur solution, un identifiant unique est donné à chaque fonction. Lors de la compilation, des instructions de vérification d'identité des fonctions appelées et point de retour sont insérées avant chaque appel et retour de fonction.

Lors de la sortie de la fonction, la vérification est faite avant le retour.

Ce mécanisme a un inconvénient principal, tous les points de retour d'une fonction vont avoir le même identifiant. Des fonctions très appelées, par exemple `malloc`, ont un grand nombre de points de retour différents dans un programme. Cela se traduit par trop d'identifiants partagés dans le programme protégé, il est donc possible de changer le flot d'exécution tout en contournant l'exécution initiale. Cela est fait en réordonnançant les *basic blocs* partageant la même vérification de retour d'une façon non prévue dans le graphe d'exécution initial. Le modèle d'exécution est trop faible.

Les gadgets protégés par leur solution ont pour instruction pivot les instructions d'appels de fonction indirects, et les retours. Les gadgets non alignés présents dans le programme protégé ne sont pas pris en compte dans cette première protection proposée.

CCFIR

Cette protection [Zha+13] est aussi une implémentation de la validité du graphe de contrôle du programme protégé. La vérification est faite par le binaire lui même. Cette protection améliore le CFI précédent. La vérification du saut est faite dans une zone mémoire dédiée, protégée en lecture face à un attaquant. Tous les sauts indirects du programme se dirigent vers cette zone mémoire et le code dans cette zone vérifie l'intégrité du saut à opérer.

Pour éviter une rétro-ingénérie du binaire, la table des sauts est générée avec un aléa au lancement du programme. Certaines propriétés, notamment de parité sur les adresses, sur la génération de cet espace mémoire doivent être respectées lors de la génération. L'implémentation proposée est basée sur des vérifications de parité, et sur la présence de bits de poids faible et poids fort sur l'adresse de saut. Lors de l'exécution d'une instruction pivot, l'adresse correspondante est vérifiée sur les tests de parités. L'exécution du programme est arrêtée si ces tests échouent.

Dans un cadre où cette zone mémoire est effectivement protégée en lecture, la protection est efficace sur les gadgets alignés. Cette protection n'est cependant pas utilisée car dans un contexte réel, la protection en lecture d'une zone mémoire n'est pas atteignable. De plus, de même que le CFI d'Abadi [Aba+09], les gadgets non-alignés présents dans le binaire ne sont pas protégés.

PICON

PICON (*Protect Integrity of CONtrol flow*) est une autre protection qui repose sur de la vérification dynamique d'intégrité du graphe de contrôle. Cette solution proposée par l'ANSSI [CFC15] est la seule qui repose sur un moniteur extérieur d'exécution. L'avantage principal d'une telle architecture est une meilleure protection, en principe, car la vérification est effectuée dans un espace mémoire complètement isolé. En contrepartie, à cause des coûts de communication entre les processus, la dégradation des performances est plus importante que sur d'autres protections.

Comme d'autres solutions de CFI, la protection est déployée à la compilation. Les appels de fonctions et *basic blocs* sont identifiés au niveau de la représentation interne du compilateur, ici LLVM. Le choix de ce compilateur permet d'être adapté à plusieurs langages pour les sources – ceux qui compilent vers LLVM – et permet d'être déployé sur plusieurs architectures matérielles.

Lors de la compilation, la solution propose de chercher à décompiler les dépendances pour protéger les appels vers des bibliothèques qui ne sont pas déjà protégées. Cela permet de faire cohabiter un programme protégé et d'autres non protégés utilisant les mêmes bibliothèques dynamiques, comme la `libc`.

La protection a cependant des limitations communes à d'autres solutions qui sont déployées au niveau de la représentation interne du compilateur, les gadgets non-alignés ne sont pas détectés. De plus les auteurs précisent que la solution ne peut pas être appliquée à des logiciels qui fonctionnent sur plusieurs *threads*.

1.7.3 GFree

Cette solution, décrite en 2013 [Ona+10], implémentée sur le compilateur clang en 2017 [4], vient compléter des solutions de CFI. La vulnérabilité traitée par cette protection est la présence de gadget non-alignés. Le principe de la protection est d'insérer des instructions `nop` dans le programme aux endroits clés pour empêcher l'interprétation de l'exécutable de manière non voulue. De plus, certaines instructions sont remplacées par une ou plusieurs autres instructions pour effectuer un calcul différent ayant le même résultat. Ces modifications permettent par exemple de faire disparaitre des `ret`.

4. https://github.com/pagabuc/gfree

Un exemple que les auteurs présentent est le suivant :

```
05 aa C3 00 00    add %eax,0xc3aa
```

devient

```
bb aa 03 00 00    mov %ebx,0x3aa
81 cb 00 c0 00 00 or %ebx,0xc000
01 d8             add %eax,%ebx
```

À la fin de l'opération, le registre `%eax` contient bien la bonne valeur (`0xc3aa`), mais l'octet `0xc3` (qui est l'*opcode* de l'instruction `return`) n'est plus présent dans la zone mémoire à un endroit non protégé.

Cette protection permet ainsi d'enlever environ 60 % des gadgets présents dans un binaire. L'outil utilisé pour récupérer la liste des gadgets n'est cependant pas renseigné.

1.7.4 Stack Ghost

Les protections proposées précédemment sont pensées et développées pour des architectures matérielles de la famille x86. D'autres protections, comme `StackGhost` [FS01], sont dédiées à l'architecture SPARC. Cette protection a la particularité d'avoir été développée avant la description académique des attaques par détournement de flot d'exécution. L'idée de la protection est de protéger la pile d'exécution contre toute corruption. Les adresses de retour sont donc, par ce biais, protégées aussi.

Contrairement à l'exécution sur x86, sur SPARC, lors d'un appel les paramètres et autres adresses ne sont pas mis sur la pile d'exécution. Un processeur de cet architecture a une fenêtre glissante sur les registres utilisés. Chaque appel et retour fait glisser cette fenêtre. Quand aucun registre n'est disponible, les registres précédents sont sauvegardés sur la pile.

`StackGhost` met en place un mécanisme de protection pour éviter la corruption des données qui ont ainsi été placées dans une zone mémoire corruptible. L'adresse des registres de retour est alors chiffrée, et contrôlée. Des instructions sont ajoutées pour qu'au retour de fonction et à la restauration des registres, les adresses soient comparées pour détecter la corruption.

Protection	Année de publication	Architecture	Besoin des sources	Mesure d'efficacité	Type de gadgets protégés	Dégradation des performances
Abadi's CFI	2010	x86	Oui : compilation	Vérification mathématique	`ret` et `call`. Alignés uniquement	0 % - 45 %
CCFIR	2013	x86	Oui : compilation	Réduction du nombre de gadgets	`ret` et `call`. Alignés uniquement	3 % - 8 %
G-Free	2013	x86	Oui : compilation	Réduction du nombre de gadgets	Tous les gadgets	0 % - 25 %
Stack Ghost	2001	SPARC	Protection matérielle	`N/A`	`N/A`	
PICON	2015	Multi	Partiel : compilation	Vérification mathématique	Gadgets alignés	3 % - 30 %

TABLE 1.1 – Récapitulatif des protections disponibles et de leur champ d'application

1.7.5 Récapitulatif

Pour résumer ces quelques protections présentées, les chiffres et éléments sont présentés dans le tableau 1.1. Les différences d'objectifs, de cible de protection, la date de publication et l'influence sur les performances y sont synthétisés. Lorsque que la protection n'a pas de chiffre relatif ou que la catégorie n'a pas lieu d'être pour la protection, le contenu de la colonne est remplacé par un `N/A`.

1.7.6 *Debloating*

Principe

Lors de l'exécution d'un programme, l'exécutable et les bibliothèques dont il dépend sont intégralement chargés en mémoire. Le principe des techniques dites de *debloating* est de réduire la quantité d'éléments chargés pour réduire la surface d'attaque. Plusieurs propositions existent pour mettre en place ce *debloating* [QPY18 ; Sha+18 ; Heo+18 ; Che+18 ; Qin+19 ; Por+20]. Qin *et. al.* [Qin+19] ont, par exemple, montré que la bibliothèque standard C liée par l'exécutable de Firefox n'est utilisée qu'à 10 %, et les 1453 binaires qu'ils ont analysés utilisent en moyenne moins de 3 % de cette bibliothèque. L'intégralité du code non utile peut être retiré, supprimant ainsi un grand nombre de gadgets disponibles.

Le *debloating* se décompose en trois étapes. La première étape est une analyse statique de l'exécutable pour déterminer quelles sont les parties non utilisées

des bibliothèques. Les deuxième et troisième étapes modifient le processus de chargement du programme. En effet, il faut modifier dans un premier temps le chargement des bibliothèques pour ne charger que ce qui est nécessaire. Ensuite, les adresses doivent être recalculées pour permettre la bonne exécution du programme.

Limitations

Une étude a montré que certaines méthodes de *debloating* ne produisent pas une augmentation de la sécurité face aux attaques par détournement de flot d'exécution. Brown *et. al.* [BP19] montrent que malgré une diminution du nombre de gadgets présents après le processus de *debloating*, il reste suffisamment de gadgets pour construire des chaînes de détournement. De plus, ils mettent en avant que les techniques de *debloating* étudiées entrainent l'introduction de gadgets qui ne sont pas présents sans utilisation de ces protections, ce qui peut être considéré comme une augmentation de la surface d'attaque.

1.8 Mesures de sécurité

La vérification de la sécurité d'un logiciel est un problème complexe. La sécurité contre les détournements de flot d'exécution n'est pas un cas particulier. Cette section présente plusieurs études qui répondent au problème de manière différente. La première étude présentée compare des solutions de CFI. Ensuite, cette section présente un outil qui permet d'améliorer des attaques existantes pour contourner des protections. Dans les deux cas un ensemble de métriques sont mises en place.

1.8.1 Sécurité des implémentations de CFI

Dans le cadre des attaques par détournement de flot d'exécution, une étude par Burow [Bur+17] propose une comparaison des différentes solutions de contrôle d'intégrité du flot d'exécution. L'étude porte sur 25 implémentations de CFI et les évaluent en terme de sécurité et de performances.

La première particularité de cette étude est la manière avec laquelle ils définissent les gadgets, nommés segments dans l'article, utilisables pour une attaque par détournement de flot d'exécution. La notion introduite est celle de classe

d'équivalence des sauts indirects. Une classe d'équivalence est l'ensemble des cibles possibles pour un transfert de contrôle indirect donné. Par exemple, en C, l'ensemble des valeurs que peut prendre un pointeur sur fonction va être la classe d'équivalence de ce pointeur lors de son déréférencement. Pour reprendre le vocabulaire de cette thèse, la classe d'équivalence d'un gadget X est l'ensemble des gadgets adressable par ce gadget X. L'analyse statique de cet espace d'adressage disponible est simple pour permettre une comparaison des différentes solutions.

Quatres axes d'étude sont utilisés pour permettre la comparaison des solutions de CFI. Le premier est le type d'instruction de contrôle du flot supporté. Certaines solutions ne protègent par exemple que les `return`. Ils identifient neuf catégories d'instructions pivots, à prendre en compte. Deux de ces catégories de pivots sont spécifiques à un langage, une pour le `C++` et une pour le `Smalltalk`.

Le deuxième axe utilisé pour la comparaison n'est pas orienté sur la sécurité mais sur les performances. Les solutions sont comparées sur la base de la dégradation des performances. Les chiffres utilisés sont à chaque fois les chiffres donnés par les auteurs des solutions de CFI.

Les deux derniers axes qui sont utilisés pour la comparaison sont deux indicateurs d'analyse statique. Ils regardent les catégories de transfert de contrôle définies dans leur article, qui sont prises en compte par l'analyse statique du binaire. Deux scores sont donnés, un pour l'analyse statique *forward*, `jump` et `call` et un pour l'analyse statique *backward*, les `return`. Par exemple, pour la précision de l'analyse statique de retour, trois niveaux sont définis :

— Aucune validation de branche retour
— Identification des classes d'équivalences
— *Shadow stack*

Les solutions sont comparées sur l'ensemble de ces quatres axes, en faisant une distinction sur la stratégie d'implémentation. En effet, trois manières de procéder sont répertoriées : instrumentation sur le binaire, utilisation d'un compilateur spécifique et protection matérielle.

Enfin une dernière métrique quantitative est définie pour comparer de manière plus fine cinq solutions de CFI qui fonctionnent sur des sources. La métrique a été définie pour essayer de palier au simple dénombrement des classes d'équivalence disponibles. En effet, ils considèrent que ce dénombrement n'est pas suffisant. Ils définissent donc un score comme étant le rapport entre le nombre de classes d'équivalence disponibles et la taille de la plus grande classe d'équivalence. Cette métrique est justifiée par le fait qu'une grande classe d'équivalence permet à un

attaquant d'avoir un plus grand choix de chemins d'attaques.

Cette approche donne des résultats concluants qui permettent de classer un type de protection contre les attaques par détournement de flot de contrôle. Les éléments mis à disposition permettent de comparer une nouvelle solution aux solutions existantes à la date de publication de cet article. Bien que cela n'ait pas été fait, une solution comme PICON [CFC15] pourrait être ajoutée à cette analyse pour être comparée à d'autres solutions.

Cependant, en dehors de la comparaison des solutions de CFI, l'approche est difficilement utilisable telle qu'elle est définie. Par exemple, `gfree` [Ona+10] est une solution qui ne peut pas être comparée aux solutions de CFI par cette méthode. C'est une solution spécifique aux architectures matérielles de la famille x86, qui apporte une sécurité importante, comme cela est montré en section 3.1.

Cela vient principalement du fait que la définition des classes d'équivalences cherche des cibles légitimes du code disponible. Le fait que seul des solutions de CFI soient considérées rend la méthode efficace : toutes les entrées et sorties de *basic bloc* sont considérées. Le principe du CFI étant de vérifier que toute sortie d'un bloc correspondent bien à l'entrée d'un autre bloc. L'intégrité d'un bloc ne pouvant pas être garantie, la présence de cibles utilisables à l'intérieur d'un bloc légitime et leur protection n'a pas d'influence sur leurs métriques de sécurité. Par exemple, la présence de gadgets non-alignés à l'intérieur d'un bloc n'est pas prise en compte.

Enfin, pour permettre de qualifier différents langages, cibles matérielles ou autres paramètres à prendre en compte lors de la conception d'un logiciel, la méthode n'est pas appropriée car elle repose sur des considérations propres aux mécanismes de défense de type CFI, et donc ne sont pas généralisables à toutes les types de protection.

1.8.2 Amélioration d'attaques

D'autres auteurs ont mis en place des outils facilitant l'attaque, notamment Q [SAB11] pour montrer que les défenses habituelles, notamment l'ASLR, ne sont pas suffisantes. Les auteurs proposent un outil permettant d'améliorer des attaques existantes sur des systèmes Linux et Windows pour qu'elles fonctionnent encore contre un système sur lequel l'ASLR et le DEP sont activés. Leur outil se concentre uniquement sur les gadgets dont l'instruction pivot est un `return`.

Le premier point mis en avant dans leur solution est la catégorisation des gadgets trouvés dans un binaire. Leur approche sur ce point diffère de ce qui est fait dans cette thèse. En effet, ils ne se basent pas sur la catégorisation des instructions données par le standard de l'architecture matérielle, mais ils se basent sur une définition très large des instructions sur la seule base de `Registre Sortie`, dans lequel (ou lesquels) l'instruction écrit, et `Registre d'entrée`, registres uniquement lus. Cela permet notamment d'étendre la catégorie `datamov` des gadgets, par exemple avec des gadgets de la catégorie `arith`. Ainsi tout gadget écrivant dans un registre une nouvelle valeur issue d'un autre registre est un gadget de catégorie move. Ils utilisent cette notion plus large de catégorie pour reconstruire des gadgets qui manquent dans un binaire pour construire une attaque.

Sur la base de cette catégorisation, les auteurs de Q trouvent dans des binaires de taille bien plus faible que précédemment tous les gadgets nécessaires à la construction d'une chaîne de détournement de flot d'exécution. Par rapport à ce qui existait avant, ils proposent plusieurs améliorations. Premièrement, leur recherche de gadget est automatisée, les premières études de gadgets étaient manuelles. Ensuite, ils montrent une efficacité de leur outil sur des programmes de plus de 20 ko, à 80 %. C'est la taille de l'intégralité du binaire qui est considérée, pas seulement la taille de la section exécutable, contrairement à ce qui est fait dans cette thèse. Ils se démarquent ainsi des recherches automatiques précédentes, qui se concentraient sur des bibliothèques exécutables de plus d'un mégaoctet comme la bibliothèque standard C ou sur un noyau Linux dont la taille dépasse plusieurs mégaoctets.

L'ensemble des gadgets considérés pour arriver à ce résultat est assez restreint. Plutôt que d'essayer de catégoriser les gadgets avec un effet de bord pour vérifier que cet effet est gênant pour la création d'une chaîne de détournement de flot d'exécution, ils sont mis de côté. L'intégralité des gadgets utilisés dans leur approchent est donc sans effet de bord. Cela réduit fortement le nombre de gadgets disponibles, bien que le chiffre de cette diminution ne soit pas connu.

La vérification de l'efficacité de leur outil pour améliorer des attaques est assez simple. Neufs attaques réelles pour des systèmes Windows et Linux qui fonctionnent sur des systèmes sans ASLR et W⊕X et qui échouent sur les mêmes systèmes protégés sont sélectionnées. Ces attaques sont améliorées avec leur outil Q et sont montrées comme fonctionnelles sur les systèmes protégées sans modification manuelle des attaques. En plus de cette évaluation, les auteurs vont vérifier si des chaînes de détournement arbitraire sont constructibles pour remplir un des trois objectifs suivants :

— Appel de fonctions appelées par le programme originel
— Appel vers des fonctions arbitraires de la bibliothèque standard C
— Écriture de 4 octets à une adresse mémoire arbitraire

Le chiffre des 80 % de réussite de création de `payload` sur les binaires de plus de 20 ko correspond à la première mesure. Pour les appels vers les fonctions de la bibliothèque C standard, le même taux de réussite est atteint pour les binaires de plus de 100 ko. Aucun chiffre n'est donné explicitement pour le troisième point.

La méthode produit des résultats montrant l'insuffisance des protections existant à la date de publication, à savoir l'ASLR et le DEP. L'efficacité de la solution sur des protections plus récentes est encore à démontrer. Cependant, certains auteurs de protections ont eu accès à l'outil pour vérifier l'efficacité des dites protections. De plus, la méthode de recherche des gadgets n'est pas vraiment précisée. Par exemple, aucune notion explicite sur les gadgets non alignés n'est avancée. L'approche présente des notions de catégorisation reprise ensuite dans la littérature. C'est le cas de l'analyse de Follner [FBB16] présentée en section 2.3.2 en page 61.Parmi d'autres outils l'un a été créé en 2014 pour contourner des défenses connues [CW14]. Un autre outil, plus récent, est proposé par Schwartz et. al [Sch+20] pour faire de la recherche de gadgets même en présence de défenses inconnues. L'outil n'a pas été étudié, dû à une publication trop récente.

1.8.3 Logiciels vétustes

Il est aussi possible de vouloir maintenir des logiciels au niveau du binaire, notamment pour des logiciels en fin de vie, ou pour des systèmes à très longue vie. Certaines dépendances ne sont pas forcément maintenues et peuvent présenter des vulnérabilités. Ali et. al [AAS20] proposent une solution pour isoler les bibliothèques dont dépend un logiciel pour continuer à les utiliser sans propager la vulnérabilité. Cependant, ils ne considèrent pas directement les attaques par détournement de flot d'exécution, car leur objectif est la protection contre les corruptions de mémoires.

1.8.4 Autres mesures de qualité

Les mesures et protections présentées dans ce chapitre sont orientées sur les attaques par détournement de flot d'exécution. Bien que cela ne soit pas discuté dans la suite du document, il existe d'autres métriques et méthodes pour mesurer la sécurité et la sureté de fonctionnement, au niveau des sources du logiciel [RH97 ;

Sta+02 ; BL05 ; Jia+08 ; RHS98 ; ANS] ou même la maintenabilité des sources d'un logiciel [Cha+14 ; Sta+02 ; VM17]. Une autre étude questionne la fiabilité des métriques vues par les développeurs les utilisant [PLB18], et montre que certaines métriques ne sont pas perçues correctement lors du développement d'un logiciel, et ne permettent donc pas facilement à un développeur d'évaluer la qualité des sources qu'il écrit.

Les travaux décrits dans cette thèse se concentrent sur la sensibilité observée des binaires, et ne concernent que très peu les sources d'un logiciel. Bien que des travaux sont faits sur l'étude des compilateurs, les métriques et méthodologies utilisées pour évaluer la qualité des sources n'ont pas été utilisées lors de l'étude de la sensibilité d'un logiciel face au détournement de flot d'exécution.

1.9 Conclusion

Un logiciel est sensible à des attaques par détournement de flot d'exécution, les binaires associés à ces logiciels peuvent être protégés, mais de manière insuffisante. Ce chapitre présente les concepts manipulés dans cette thèse permettant d'arriver à cet état des lieux. Un logiciel a été défini, et les différents modèles d'exécution ont été présentés. Ces descriptions ont permis d'expliquer le fonctionnement des attaques par détournement d'exécution. Ce chapitre définit aussi ce que sont les gadgets, éléments qui constituent ces attaques. La construction des attaques et des moyens de s'en prévenir ont été présentés ainsi que les limitations induites.

Ce chapitre nous a permis de voir que de nombreuses protections sont disponibles pour protéger un logiciel contre du détournement de flot d'exécution. Un point qui ressort de cette présentation est qu'il est difficile de construire des éléments de comparaison qui permettent à la conception d'un logiciel de faire des choix éclairés propres à des besoins spécifiques. Bien que la plupart des solutions proposent des chiffres comparables pour la perte de performance, les chiffres donnés pour le gain de sécurité sont difficilement quantifiable.

De plus, les protections présentées apparaissent tardivement dans le processus de conception d'un logiciel. Des protections vont être déployées directement sur un binaire produit, comme [Aba+09], d'autres nécessitent un compilateur particulier, comme Gfree [Ona+10]. Lors de la conception d'un logiciel, il est donc difficile d'évaluer précisément l'interaction que peuvent avoir deux protections, ou s'il peut être intéressant de porter des principes spécifiques d'une protection d'un

compilateur à un autre.

Le point qui nous manque pour étudier l'impact des différentes étapes de la conception sur la sensibilité d'un logiciel est une ou plusieurs métriques permettant de facilement comparer un grand nombre de logiciels. L'intervention humaine n'est pas une option qui permet de facilement passer à l'échelle. De même, comme montré pour les améliorations d'attaques de l'outil Q en section 1.8.2, l'utilisation d'attaques existantes sur des logiciels définis, souvent des *Common Vulnerabilities and Exposures (CVE)*[5], n'est pas idéale. En effet, cette approche est trop réactive et ne mesure l'efficacité d'une protection que contre des attaques spécifiques et connues.

Dans cette thèse, plusieurs éléments sont proposés pour palier à ces manques. Le chapitre 2 présente les différentes métriques qui sont mises en place pour comparer les différents éléments qui influent sur la sensibilité potentielle d'un logiciel face aux attaques par détournement de flot d'exécution. Dans ce chapitre, des éléments mis en avant ici sont réutilisés pour permettre de confirmer les résultats présents dans la littérature. L'évaluation des métriques et leur utilisation dans la conception amont d'un logiciel sont faites respectivement dans les chapitres 3 et 4.

5. https://cve.mitre.org/

Chapitre 2

Analyse et mise en place de métriques

2.1 Analyse d'un corpus existant

La conception d'un système utilise souvent des éléments quantitatifs et qualitatifs permettant de comparer sur différents axes les solutions à disposition. Les diverses contraintes des systèmes en matière de performances impliqueront des choix d'architecture matérielle, de langages ou de bibliothèques différents. Lors de l'étude des performances d'un logiciel, plusieurs suites de logiciels de test permettant d'évaluer l'apport d'une technologie ou d'une évolution sont disponibles. On trouve par exemple le SPEC CPU2000 [Hen00], son évolution SPEC CPU2006 [Hen06]. Ces deux suites sont assez populaires et servent d'étalon pour comparer deux solutions en termes de performance.

D'un autre côté, pour la sécurité d'une application, ce type de suites logicielles n'existe pas pour le moment. Nous avons vu au chapitre précédent que la mesure d'efficacité d'une protection est souvent faite au cas par cas par les créateurs de protections, comme explicité en section 1.8. Des suites de *benchmarks* non dédiées à la sécurité sont parfois utilisées. De manière générale, les outils et méthodes utilisés sont bien différents d'une protection à l'autre.

La métrique communément utilisée pour déterminer les efforts de production réalisés est le dénombrement des gadgets dans un logiciel protégé. La manière avec laquelle ce dénombrement est effectué varie d'une protection à une autre. Deux outils émergent, le premier est ROPgadget [Sal12], le second est Q [SAB11]. Certains préfèrent utiliser des solutions internes, parfois manuelles. Le point principal qui distingue les méthodes d'évaluation est l'ensemble de binaires sur lequel les protections sont évaluées. Certains appliquent leurs mesures sur des logiciels pour

leur exposition à des attaquants, comme `nginx` ou `zip`. Il est fréquent que seuls des ensembles limités soient testés.

Pour comparer différents outils utilisés lors du processus de conception d'un logiciel sécurisé, il fallait donc étendre ce qui existe, comme par exemple [Bur+17] pour la comparaison des techniques de CFI. Le but est d'avoir une méthodologie et des métriques uniformisées permettant de facilement passer à l'échelle. L'intervention humaine doit être limitée au maximum. Une métrique à éviter, par exemple, est « nous n'arrivons pas à construire une chaîne d'attaque sur les binaires protégés ». Elle peut donner un indice dans un contexte particulier, mais n'aide pas forcément à bâtir une confiance dans la protection fournie, et ne passe pas à l'échelle également.

Ce chapitre et le suivant, en page 71, montrent que le processus de compilation est complexe. La comparaison de différents processus et des paramètres qui influent dessus de manière exhaustive est jugée trop coûteuse à mettre en œuvre. Le choix a été fait de partir sur une analyse des binaires pour déterminer la sensibilité d'un logiciel. Un logiciel va donc présenter plusieurs sensibilités aux attaques par détournement de flot d'exécution en fonction des binaires choisis. Cette approche présente un avantage présenté en section 2.2.1, la disponibilité des binaires sur des distributions GNU/Linux. Cette approche permet aussi d'étendre le champ d'études assez rapidement, puisque seuls des binaires, avec une description de leur provenance, est nécessaire pour améliorer la pertinence statistique de l'étude. Cette approche statistique, développée dans ce chapitre, sur l'étude des binaires permet de cerner des métriques de sensibilité utiles.

2.2 Mise en œuvre

2.2.1 Qualification d'un corpus d'analyse

Pour mettre en place des métriques de sécurité contre le détournement de flot de contrôle, il faut obtenir un ensemble de logiciels variés à analyser. Cet ensemble doit avoir plusieurs propriétés importantes. En effet, l'ensemble doit être suffisamment varié pour éviter d'avoir des biais si les systèmes analysés sont trop spécialisés. D'un point de vue pratique, les logiciels sont analysés au travers du binaire final. La sécurité d'un logiciel va dépendre de plusieurs paramètres discutés au chapitre 3. Les logiciels sont associés à leurs binaires respectifs dans chacun des

systèmes où ils sont présents. Ne se concentrer que sur des systèmes embarqués pourrait empêcher d'avoir affaire à des binaires qui ne sont pas optimisés pour la place prise sur le disque. C'est une optimisation qui a un effet assez fort sur la réutilisation de code d'un binaire et qui peut avoir un effet sur les dépendances du logiciel.

Pour éviter de n'avoir qu'une seule catégorie de système décrite en section 1.2.2, page 11 dans notre corpus d'analyse, la provenance des binaires est diversifiée au maximum. Ainsi, on trouve dans notre corpus aussi bien des binaires issus de systèmes embarqués que de systèmes plus classiques, serveur et bureau notamment.

Le point suivant qui a été considéré lors de la construction du corpus de binaires porte sur les objectifs principaux du système. En effet, entre un serveur de services web, un ordinateur de bureau ou même un ordinateur de bord d'une voiture peuvent contenir des logiciels, configurations et paramétrages bien différents. Les logiciels peuvent être construits avec des objectifs différents, et les implémentations d'un logiciel peuvent aussi varier, par exemple pour avoir une version plus légère en espace disque. Il est aussi concevable qu'un logiciel soit écrit et compilé avec les mêmes options sur deux systèmes, mais qu'ils utilisent des dépendances différentes. Un logiciel peut opter lors de la phase de production de dépendre de la bibliothèque `musl` plutôt que la `GNU libc`, qui fournit les mêmes services. Le choix peut être motivé pour des raisons de sécurité, de licence, de disponibilité ou juste de préférence. D'un système à l'autre, le type de logiciels présents peut aussi varier. Par exemple, il est fort peu probable de trouver des logiciels graphiques sur un serveur, alors qu'il est rare d'utiliser un ordinateur de bureau qui n'en a aucun.

Le dernier critère pris en compte lors de la construction du corpus est l'architecture matérielle cible du système. Les architectures choisies sont en lien avec le type de systèmes, certaines architectures sont notamment très populaires pour les logiciels embarqués. L'architecture ARM est prédominante sur toute la téléphonie. Dans le domaine de l'aviation, un certain nombre de systèmes utilisent des processeurs dont l'architecture peut être SPARC ou encore PowerPC. Une architecture matérielle a des procédures d'appel de fonction et de contrôle de flot d'exécution qui vont changer fortement les gadgets qui se trouvent dans un binaire.

Enfin, plutôt que de construire plusieurs systèmes pour faire varier ces paramètres, les binaires collectés proviennent de systèmes existants. Cela présente plusieurs avantages. Le premier est d'éviter les biais pour déterminer les logiciels que le système doit avoir pour remplir les objectifs de sa mission. Par exemple, cela peut éviter d'avoir à faire des hypothèses du type « un système embarqué

fournissant des services webs utilise forcément `lighttpd` plutôt que `apache` ≫. Cette affirmation peut sembler valide, du fait des ressources utilisées par les deux logiciels, mais elle reste biaisée. Cependant, certains systèmes ont été créés de toute pièce, pour cause d'absence de données. Une synthèse de l'ensemble des systèmes sur lesquels les binaires ont été collectés est décrite dans la première partie du tableau 2.1.

Tous les systèmes considérés sont des `GNU/Linux`. L'inclusion d'autres systèmes d'exploitation n'est pas exclue, mais n'a pas été intégrée dans les travaux présents. Sur ces systèmes `GNU/Linux`, seuls les binaires présents dans le dossier système `/usr/bin` ont été collectés. Bien que cela ne représente pas l'intégralité des logiciels présents sur ces systèmes, la majeure partie des binaires y sont. Par exemple, à l'exception du noyau et d'une dizaine de logiciels particuliers, sur le système Arch Linux analysé, tous les binaires systèmes y sont installés.

Ce tableau contient aussi des ensembles de binaires qui ne proviennent pas de systèmes en particulier, et qui ont été collectés pour affiner des analyses qui seront discutées dans le chapitre 3. Leur inclusion est motivée par différentes raisons. Deux ensembles, `tar` et `Cometbuster`, sont intégrés pour étudier l'influence du processus de compilation sur la sécurité d'un logiciel. Ensuite vient l'ensemble nommé `Opam` qui est constitué d'un ensemble de binaires écrits en OCaml, ensemble qui a été ajouté pour étudier l'influence du langage. Enfin, l'ensemble des binaires distribués par Mozilla du navigateur web `Firefox`, en 64 bits, de la version 4.0 à la dernière version disponible à la date de la collecte. Ce dernier ensemble a été ajouté afin d'étudier l'évolution dans le temps de la sensibilité d'un logiciel.

Nom du système	Date de *snapshot*	Cible	32/64 bits	Architecture	Nb binaires	Notes
Fedora 26	2018-01-19	Bureau	64	x86_64	1615	
Ubuntu 16.04	2017-10-10	Bureau	64	x86 64	2019	Système riche : cible éducative
Debian Testing	2018-01-15	Bureau	64	x86_64	1001	
Debian 9.3	2018-01-12	Serveur	64	x86_64	310	
Gentoo Linux	2017-09-07	Serveur	64	x86_64	739	
Arch Linux	2017-11-23	Bureau	64	x86_64	1947	
Buildroot	2018-02-05	Embarqué	32	ARM	115	
=	=	=	64	ARM	114	
=	=	=	32	SPARC	89	
=	=	=	32	SPARC	89	Dérivée "mineure" de SPARCv8
=	=	=	64	x86_64	110	
=	=	=	32	i386	111	
Debian 9.3	2018=10=10	Embarqué	64	x86_64	199	
=	=	=	32	i386	199	
=	=	=	64	ARM	199	
=	=	=	32	ARM	199	ARM EABI
=	=	=	32	ARM	199	ARM *hard float*
Tar	release v1.30	N/A	64	x86_64	407	Compilateurs et options variables
Firefox	2019-01-30	N/A	64	x86_64	829	Toutes les versions de la 4.0 à la 65.0b9
Cometbuster	2018-02-13	N/A	64	x86_64	357	Jeu graphique, variation d'options de compilation
Opam	2019-05-13	N/A	64	x86_64	222	Ensemble de binaires OCaml distribué via opam
Total					11 179	

TABLE 2.1 – Provenance des binaires analysés

2.2.2 Outillage

Ce corpus a été construit et analysé en plusieurs étapes. Cette section décrit les procédés utilisés et l'architecture du système mis en place pour analyser les binaires des différents systèmes présentés. Le système d'analyse est composé de deux étapes principales indépendantes l'une de l'autre. La première est la partie collecte de données, la seconde l'analyse en elle-même.

La première étape de cette analyse de sensibilité des binaires est la création d'une base de connaissance regroupant les informations nécessaires. Cette base de connaissance regroupe des informations de différentes natures :

- Les informations concernant le système de provenance
- Pour chacun des binaires, les informations génériques suivantes :
  - Architecture (ARM, MIPS, x86,...)
  - Mode (32, 64 bits)
  - Format (ELF, PE, MACHO)
  - Boutisme (gros-boutiste, petit-boutiste) ou *endianess* en anglais
  - Taille du binaire
  - Taille des sections exécutables
- Tous les gadgets associés à un binaire ainsi que :
  - Le nombre d'apparitions de chacun des gadgets
  - La catégorisation de chaque gadget
  - Le score d'utilisabilité de chaque gadget (expliqué en section 2.3.2)
  - La forme canonique de chaque gadget (expliquée en section 2.3.3)

Pour la réalisation de cette base de connaissance, l'outil choisi est `MariaDB` pour le stockage. Le schéma de la base de données est présenté en figure 2.1. Le peuplement de la base de données est réalisé avec plusieurs outils. Pour les informations générales concernant les binaires, un outil a été développé en python en se basant sur la bibliothèque `Capstone`. Les gadgets de chaque binaire ont été obtenus avec un outil *open source* disponible sur `github` : `ROPgadget` [Sal12]. Bien qu'un autre outil soit disponible sur demande, Q [SAB11], il n'a pas été retenu, notamment car il est plus difficile d'échanger ou de distribuer l'outillage complet avec des industriels impliqués dans la thèse. `ROPgadget` présente les avantages suivants qui justifient le choix par rapport à un développement d'outil interne :

- Facilement déployable. L'outil en python a peu de dépendances (Seulement la bibliothèque `Capstone`)
- Supporte toutes les principales architectures matérielles (ARM, Aarch64, SPARCv8, i386, x86_64,...)

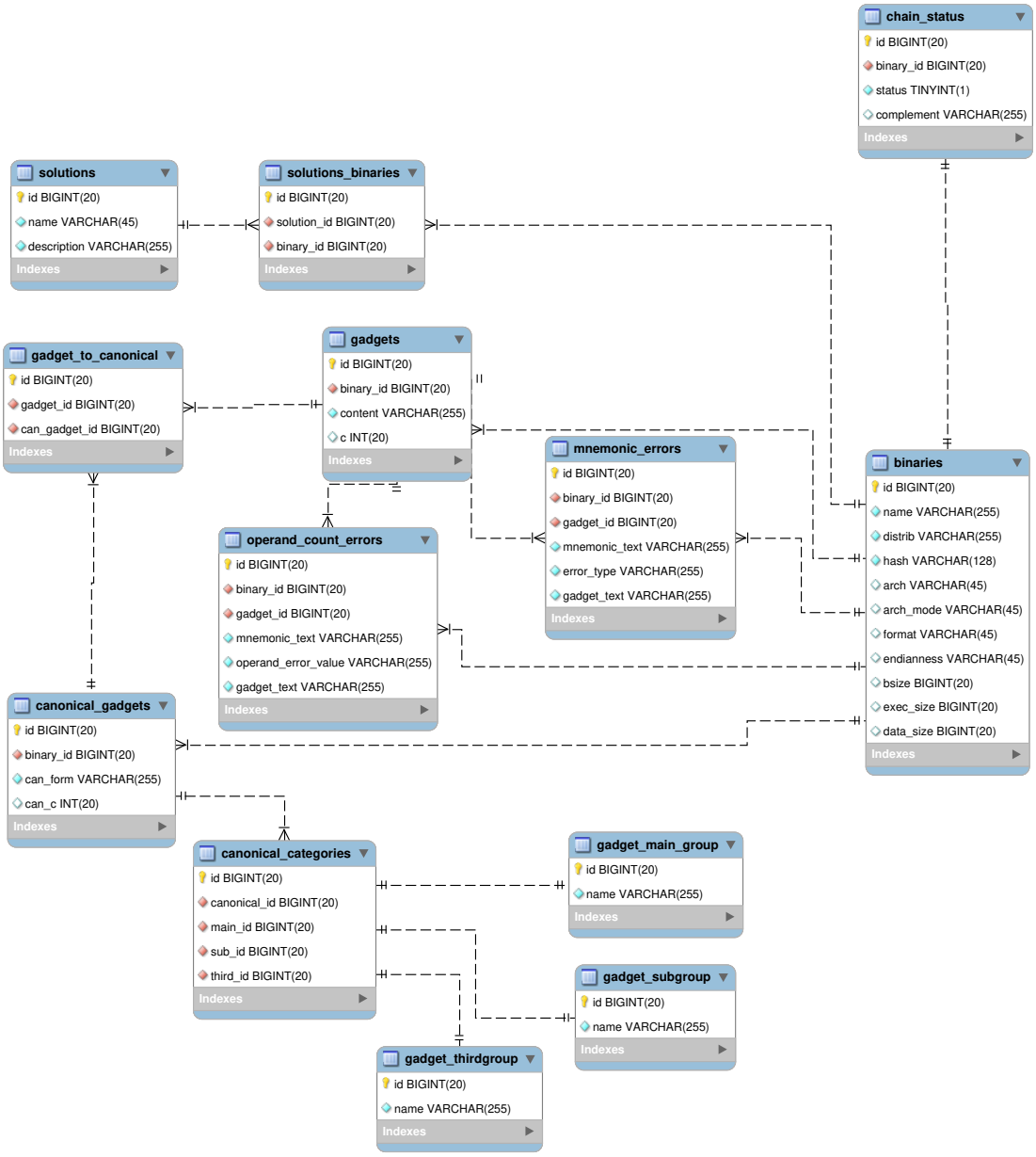


FIGURE 2.1 – Diagramme de la base de données contenant les informations des binaires, et de leurs gadgets

— Renvoie la liste des gadgets sous forme de chaîne de caractères en assembleur associé à l'architecture analysée.

L'outil a cependant l'inconvénient de ne pas avoir de formalisme générique permettant une analyse plus générique indépendante de l'architecture. Une bibliothèque, BAP [1], sur laquelle est basée Q [SAB11], permet d'avoir une représentation des gadgets indépendante de l'architecture, mais n'a pas été sélectionnée car le temps de développement d'un outil basé dessus a été jugé trop important initialement.

La seconde étape du système d'analyse de sensibilité n'est pas déployée directement sur la base de données. Le choix a été fait de pouvoir réaliser les analyses sans accès à la base de données. Un système d'export partiel vers du `CSV` est disponible pour obtenir des fichiers pour des analyses particulières. Ce système permet d'avoir une certaine flexibilité, notamment de permettre le suivi versionné et distribué des analyses qui sont présentées dans les sections et chapitres suivants.

2.3 Métriques de protection

Lors de l'analyse des binaires plusieurs métriques ont été mises en place pour qualifier la vulnérabilité apparente d'un binaire vis-à-vis du détournement de flot de contrôle. Cette section présente des métriques quantitatives simples et certaines portants sur la qualité d'un binaire. Cette section présente l'extension de métriques déjà présentes dans la littérature et qualifie en quoi ces métriques sont encore insuffisantes aux besoins.

Plusieurs types de métriques sont possibles. Les premières qui seront décrites sont quantitatives. Pour ces métriques on s'intéresse principalement au nombre de gadgets que contient un binaire. En fonction de différents paramètres, la validité d'un simple dénombrement pour caractériser un logiciel vis-à-vis de ces attaques est étudiée, et cela est réalisé sur l'ensemble du corpus.

Pour les métriques qualitatives, seulement les binaires ayant pour cible les architectures i386 et x86_64 seront étudiés. La plupart de nos binaires (10 175 sur 11 179) sont issus de ces architectures. Ces métriques sont développées suite à la catégorisation de l'ensemble des instructions de ces architectures. Les métriques mises en place en section 2.3.2 pourraient cependant être adaptées à d'autres architectures matérielles, en adaptant la catégorisation proposée dans cette section.

1. https://github.com/BinaryAnalysisPlatform/bap

2.3.1 Métriques quantitatives

Mesure du nombre de gadgets

Avant de commencer à dénombrer les gadgets présents dans un binaire, il faut préciser ce qu'est un gadget. Un gadget est défini en section 1.4.1, en page 19, comme une suite d'instructions exécutables légitimes précédant une instruction de modification du flot d'exécution. Cette définition est précisée ici car plusieurs éléments sont à prendre en compte.

Premièrement, il faut définir le nombre d'instructions contenues dans le gadget. Dans notre cas, nous avons considéré des gadgets de 5 instructions maximum, en comptant l'instruction pivot. Ce chiffre correspond à ce qui est décrit pour les gadgets présents dans la littérature, par exemple dans [Roe+12 ; SAB11 ; Gök+14]. Les gadgets ainsi choisis offrent une surface d'analyse intéressante et permettent généralement de construire une chaîne de détournement de flot. D'un point de vue purement pratique, les mesures mises en place pourraient être étendues, il faudra compter sur une augmentation du temps nécessaire pour réaliser la recherche des gadgets et les analyses qui suivent. Plus le gadget est long, plus il y aura des instructions susceptibles de gêner l'attaquant avec des effets de bord non voulus.

Deuxièmement, il faut prendre en compte les gadgets – plus courts – qui sont inclus dans le gadget initial. Prenons par exemple le gadget de 5 instructions suivant :

```
add %rsp, 0x10 ; pop %rbx ; pop %rbp ; pop %r12 ; ret
```

Ce gadget contient 4 gadgets supplémentaires suivants :

1. `pop %rbx ; pop %rbp ; pop %r12 ; ret`
2. `pop %rbp ; pop %r12 ; ret`
3. `pop %r12 ; ret`
4. (`ret`)

Bien que le gadget ne contenant que `ret` ne soit pas utile, d'autres instructions de saut peuvent servir à reconstruire des gadgets inexistants dans le binaire, l'instruction `syscall` par exemple. En effet, ce pivot peut nécessiter un certain nombre de paramètres. Les valeurs des paramètres ont besoin d'être dans des registres précis. Un attaquant peut mettre en place tous les registres avec plusieurs gadgets quelconques et finir la chaîne avec un gadget ne contenant que le pivot `syscall`.

Ensuite, la manière avec laquelle les gadgets peuvent être cherchés peut aussi influer. Le code légitime est la zone mémoire marquée comme exécutable dans l'espace mémoire alloué au programme. Certaines architectures matérielles permettent de lire à n'importe quelle adresse dans cette zone pour trouver les instructions à exécuter, notamment i386 et x86_64, comme présenté au chapitre précédent en section 1.4.3. Les autres architectures présentes dans le corpus de binaires ne sont pas concernées par cette lecture décalée.

L'unicité d'un gadget est uniquement basée sur les mnémoniques des instructions qui le composent. Par exemple `pop %rdi; call %rdi` et `pop %edi; call %rdi` sont deux gadgets ayant une capacité de calcul similaire : ils font la même opération sur deux registres différents. Ils sont cependant considérés distincts. Le registre peut avoir une importance, justifiant cette approche. La capacité de calcul des gadgets présents dans un binaire peut varier d'un registre à un autre. Par exemple, une chaîne qui nécessite une valeur dans \`%rbx`, va avoir besoin des gadgets utilisant des registres précis pour arriver à ce but. La distinction des gadgets en fonction du registre utilisé est donc nécessaire.

Résultats généraux

Plusieurs éléments sont intéressants à étudier dans un binaire pour déterminer ce qui peut influencer la sensibilité. La première métrique mise en place est le dénombrement des gadgets présents dans un binaire, et ce de matière suffisamment simple pour donner une première mesure utilisable. Cette métrique peut déjà servir à étudier l'influence de la taille d'un binaire sur sa sensibilité. Dans un premier temps, deux mesures sont utilisées pour déterminer cette influence : le décompte des gadgets uniques dans un binaire et le décompte de l'ensemble des gadgets. La figure 2.2 montre les résultats obtenus. Chaque binaire est représenté par deux points sur la figure :

— un point bleu pour le nombre de gadgets totaux dans le binaire
— un point orange pour le nombre de gadgets uniques dans le binaire

Cette figure peut être séparée en deux parties. En effet, en dessous de 3 ko, le nombre de gadgets ne dépend que très peu de la taille de section exécutable du binaire, avec un nombre de gadgets en dessous de la centaine. Au-dessus de cette valeur de taille de section exécutable, les nombres de gadgets uniques et totaux sont corrélés à la taille de la section exécutable du binaire. Le nombre de gadgets introduits par ko de section exécutable est d'environ 30 pour le nombre total de gadgets et

d'environ 20 pour le nombre de gadgets uniques. Le point intéressant de cette figure est tout d'abord qu'il ne semble pas y avoir de limite au nombre de gadgets uniques dans un binaire. L'ajout de code exécutable à un logiciel entraine l'introduction de nouveaux gadgets, contrairement à ce que l'on pourrait initialement penser.

Le nombre de gadgets augmentant avec la taille de la section exécutable du binaire, le choix de gadgets disponibles est aussi plus varié. Certains binaires se démarquent en termes de dénombrement de gadgets disponibles. En effet, la figure 2.2 présente certains binaires très éloignés du reste, avec un nombre de gadgets jusqu'à 10 fois la moyenne à taille équivalente. Malgré un plus grand nombre de gadgets disponibles, il ressort surtout une augmentation de la duplication de gadgets dans les binaires. En effet, bien que le nombre de gadgets uniques augmente, dans le cas général, le ratio entre le nombre de gadgets total et le nombre de gadgets uniques augment avec la taille de la section exécutable. Cette observation est précisée en figure 2.3, qui présente le rapport entre le nombre de gadgets total et le nombre de gadgets uniques dans un binaire.

Cette figure montre en premier lieu, en orange, qu'un certain nombre de binaires ont un ratio de gadgets légèrement supérieur à 1, indépendamment de la taille des sections exécutables du binaire. Ces binaires sont construits pour des cibles matérielles RISC. Cette séparation peut être faite car certains binaires des architectures i386 et x86_64, en bleu sur la figure, sont construits pour des systèmes embarqués et aucun binaire, pour les tailles supérieures à 10 Ko, de ces architectures n'approchent un si faible ratio de duplication de gadgets. Quelques binaires semblent contredire la généralisation à toutes les architectures RISC, cela est discuté plus précisément au chapitre 3.

Ensuite, un ensemble non négligeable de binaires ont des ratios assez élevés. 24 binaires ont un ratio de duplication supérieur à 6, soit environ trois fois la moyenne de 2,2. Certaines valeurs extrêmes sont présentes, notamment `gregorio` qui a un ratio de duplication égal à 8,89. Ce binaire a 189 Ko de section exécutable, permet de convertir des fichiers de partition de chant grégorien en ASCII vers du TeX. Installé sur un système bureau, une Arch Linux, il est le seul binaire dans ce cas, et ne semble pas avoir de particularités qui le feraient sortir du lot. Parmi les autres binaires analysés avec un fort taux de duplication de gadgets, `Quasselcore` présente un ratio de 6,2, lui aussi construit pour une architecture x86_64, et reconstruit sur mesure pour le système sur lequel il est déployé, une Gentoo Linux.

Ces deux cas extrêmes sont cependant peu parlants car ils n'ont été trouvés

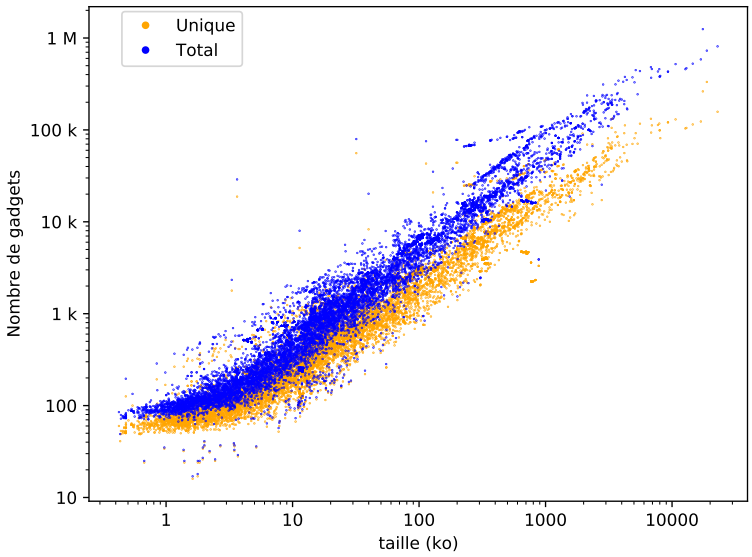


FIGURE 2.2 – Répartition des gadgets uniques et totaux en fonction de la taille des sections exécutables. 11 179 binaires

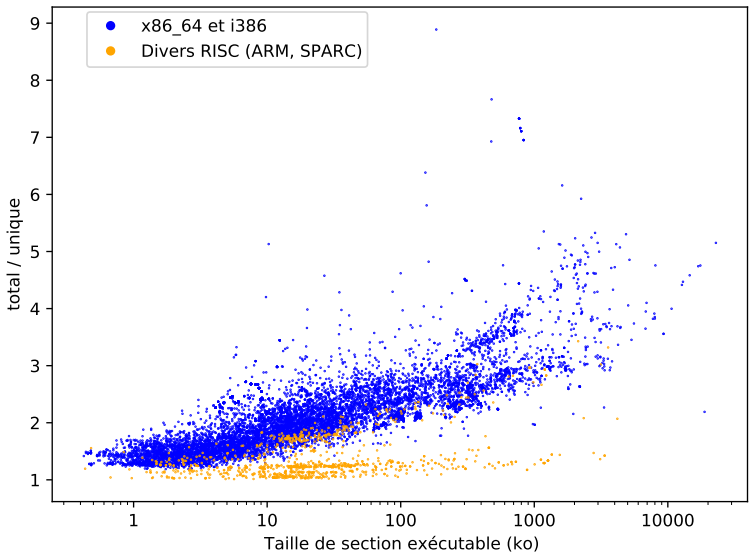


FIGURE 2.3 – Proportion de gadgets uniques par binaire en fonction de la taille de la section exécutable

que sur un système. Le cas de `screen` est plus intéressant pour voir la diversité qu'un même logiciel peut avoir d'un système à un autre. On trouve ce logiciel sur tous nos systèmes GNU/Linux et sur toutes les architectures analysées. Ce logiciel à un ratio de duplication allant d'environ 1,2 sur les architectures RISC SPARCv8, SPARC leon et ARM32. L'architecture RISC ARM64 s'écarte de la tendance de ce type d'architecture avec un ratio aux alentours de 2,1. Sur les architectures de la famille x86, ce ratio monte jusqu'à 2,5. Le tableau 2.2 présente les résultats détaillés.

Bien que cette mesure ne permette pas d'en dire beaucoup sur la sensibilité d'un logiciel, elles permettent de comparer des logiciels sur la diversité de leur surface d'attaque. En effet, la possibilité ou non pour un attaquant d'avoir plusieurs choix pour un même gadget peut permettre d'éviter par exemple d'avoir à travailler pour utiliser des adresses contenant des octets `0x00`. La raison pour laquelle cet octet n'est pas utilisé est qu'un point d'entrée fréquemment exploité est le dépassement de tampon des chaînes de caractères. Le caractère de fin de chaîne est le caractère NULL, écrit `0x00` en ASCII. Le dépassement de tampon de chaîne de caractères s'arrête donc à l'écriture d'un tel octet, limitant la corruption à ce qui précède cet octet. C'est pourquoi les chaînes évitent de recourir à des adresses ayant un octet nul.

Plusieurs catégories de gadgets ont été définies en section 1.4.2. Pour chacune de ses catégories, l'influence de la taille reste similaire. Le nombre de gadgets de chaque catégorie augmente avec la taille de la section exécutable, comme montré pour en figure 2.4. Quelques catégories ont des comportements singuliers. La catégorie `bit` par exemple, a très peu de gadgets et ne présente donc pas d'augmentation

Distribution/système	Taille de section exécutable	Nb total de gadgets	Nb de gadgets uniques	Ratio
Arch Linux (x86_64)	330 ko	13 238	5288	2,503
Ubuntu (x86_64)	319 ko	11 991	4743	2,528
Fedora (x86_64)	333 ko	13 728	5484	2,503
Debian testing (x86_64)	329 ko	13 423	5410	2,483
Buildroot (x86_64)	241 ko	9843	3995	2,476
Buildroot (i386)	240 ko	13 325	5484	2,334
Buildroot (leon)	300 ko	14 190	11 529	1,231
Buildroot (SPARCv8)	300 ko	14 220	11 547	1,221
Buildroot (ARM 32)	265 ko	8508	7135	1,192
Buildroot (ARM 64)	259 ko	16 717	7950	2,103

TABLE 2.2 – Comparaison des différents mesures et éléments du programme `screen` sur différents systèmes.

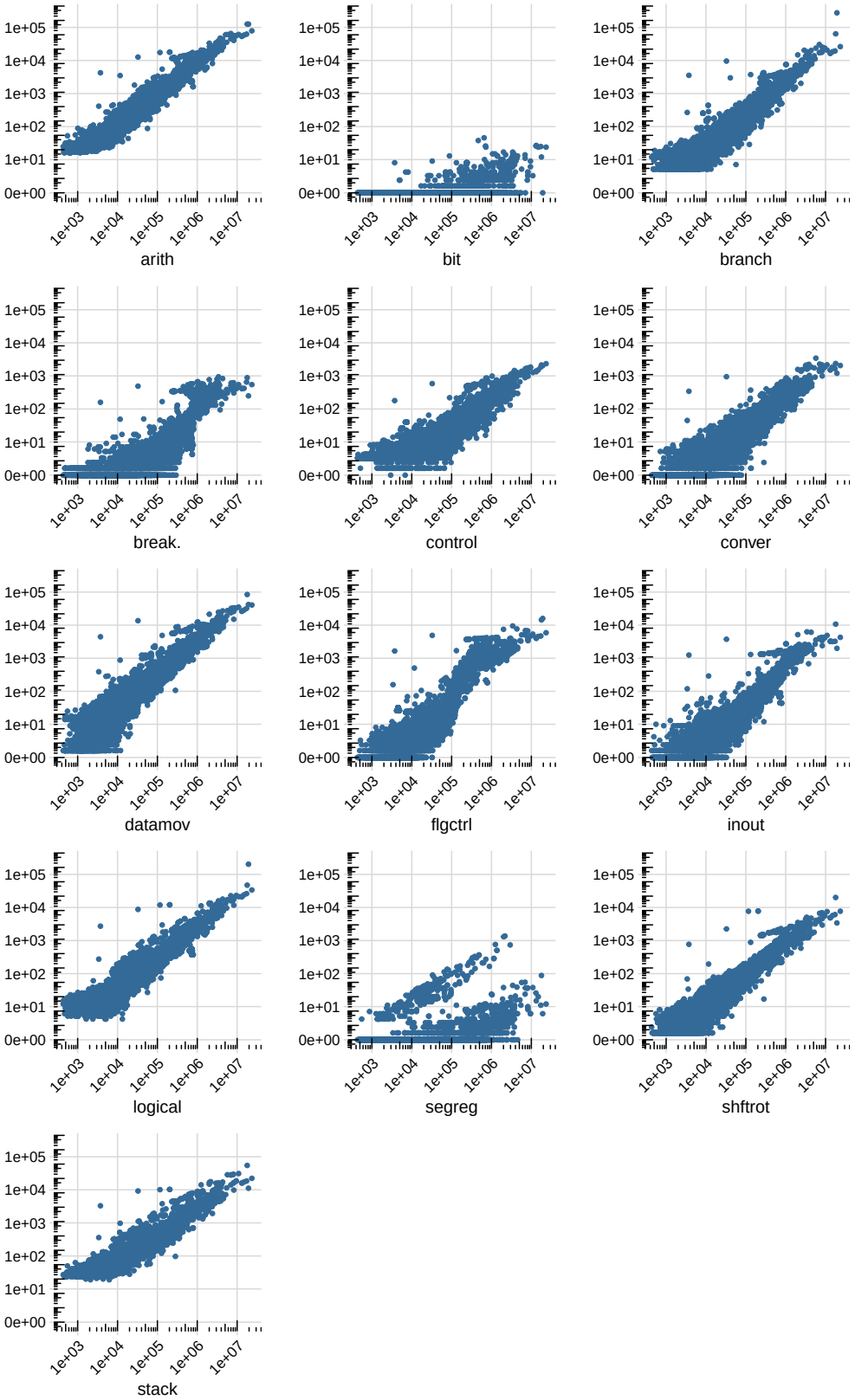


FIGURE 2.4 – Influence de la taille de la section exécutable en octet sur le nombre de gadgets, classé par catégorie de gadget

stable avec la taille de section exécutable. Cette catégorie fournit quand même plus de gadgets pour les binaires de taille importante (supérieure à 10 ko). Ensuite, la catégorie qui présente un sous ensemble de binaires ayant un nombre constant de gadgets dans la catégorie en question est `break`. En effet, pour les binaires ayant une section exécutable supérieure à 100 ko, avec environ 1000 gadgets dans la catégorie. Enfin, bien que l'explication ne soit pas connue, la catégorie `segreg` présente deux courbes bien distinctes. La catégorie étant fortement liée à l'architecture 32 bits i386, les instructions associées ayant presque complètement disparu, au moins dans l'usage, en x86_64, l'explication peut venir de là, bien que cela n'a pas été confirmé.

Pour pouvoir comparer deux binaires ayant des tailles de section exécutable différentes, une autre métrique est introduite car comparer deux binaires de tailles différentes sur leur seul nombre de gadgets n'est pas pertinent. De manière assez naïve, la métrique mise en place est une densité de gadget par kilo octet de section exécutable. Une normalisation simple à mettre en œuvre qui, a priori, pourrait être concluante. Une telle métrique permet de mieux comparer des binaires indépendamment de leur taille de section exécutable. La figure 2.5 présente les résultats sur l'ensemble du corpus de cette métrique. Cette figure montre que des densités faibles sont très peu présentes sur les petits binaires, dont la taille de section exécutable est de l'ordre du kilo octet ou en dessous. Pour les binaires au-delàs du kilo octet, la densité de gadget est bornée dans une bande, de 30 à 200 gadgets par kilo octet. Certains binaires se démarquent par leur très forte ou très faible densité de gadget, comme `gregorio`, déjà cité pour son taux de duplication, avec une densité de gadget de 200 gadgets par kilo octet de section exécutable. Dans les binaires un peu plus connus, `apt` le gestionnaire de paquet de la distribution GNU/Linux Debian a une densité de 238 par kilo octet de section exécutable. 71 binaires ont une densité supérieure à 200 gadgets par kilo octet, avec un maximum pour `gtk3-demo-application` qui culmine à 7931,72 de densité. De l'autre côté, 73 binaires ont moins de 10 de densité de gadgets par kilo octet de section exécutable, avec par exemple `perl`, interpréteur du langage du même nom, qui atteint le minimum observé de densité (4.41). On trouve à des valeurs un peu plus élevées `dpkg` (7.58) ou encore `make` (8.87). Ces faibles densités sont toutes présentes sur les deux systèmes embarqués Debian ayant pour architecture ARM, en 32 bits.

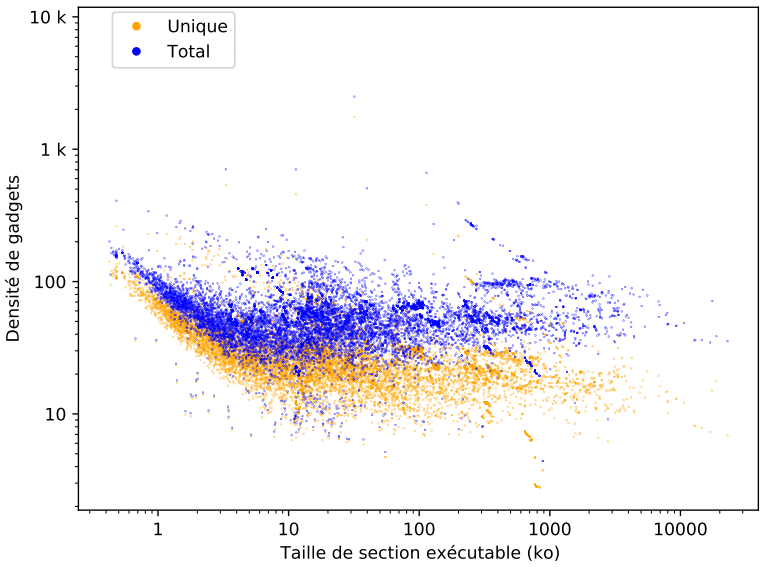
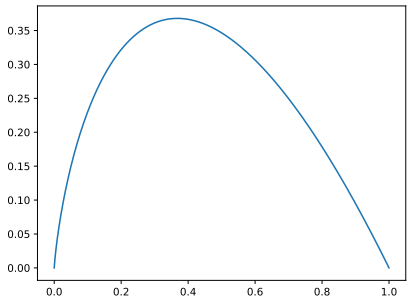


FIGURE 2.5 – Densité de gadget par kilo-octet de section exécutable, sur l'ensemble du corpus

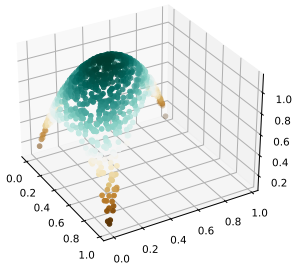
Diversité des gadgets

Pour aller plus loin dans la qualification quantitative des gadgets des binaires, cette section introduit la notion d'entropie. L'entropie est ici calculée par catégorie pour mesurer la diversité des gadgets au sein d'une catégorie. Pour cela, l'entropie de Shannon est assez utilisée dans des domaines autres que l'informatique, par exemple en géographie, pour mesurer la disparité de répartition d'une population par exemple [MO81]. La notion d'entropie au sens de Shannon est cependant souvent montrée comme insuffisante dans le domaine de la géographie ou de l'écologie par exemple [Bat74 ; PTV09 ; Jos06]. L'idée est d'avoir un indice sur la prédominance d'un gadget au sein d'une catégorie. Bien que l'indice n'est pas le plus complet, la démarche était d'essayer de valider si une telle métrique a du sens pour l'évaluation de la diversité. Cette notion est plus intéressante sur les architectures CISC, car la section précédente a montré que les architectures RISC ont tendance à n'avoir que très peu de gadgets dupliqués.

L'entropie du binaire est définie par la formule 2.1, où n_i représente le nombre d'occurrences d'un gadget dans la catégorie et n_t le nombre total de gadgets dans la même catégorie ($n_t = \sum_i n_i$). L'entropie générale d'un binaire est calculée en considérant que tous les gadgets sont dans une catégorie unique.



(a) Représentation de la fonction $y = -xlog(x)$ dans l'intervalle [0 ;1]



(b) Entropies pour différentes valeurs de répartition de trois ensembles. En abscisse et ordonnée sont les deux premiers ensembles. Le troisième est linéairement dépendant des deux autres : $c = 1 - b - a$

FIGURE 2.6 – Visualisation de l'entropie

$$S = \sum_i \frac{n_i}{n_t} \times log(\frac{n_i}{n_t}) \tag{2.1}$$

Pour commencer, la figure 2.6a présente la courbe $y = -x\,log(x)$ dans son intervalle de définition [0 ;1]. Cette fonction présente un maximum en $1/e$ et vaut $1/e$ (soit environ 0.37). La figure 2.6b présente quant à elle l'entropie pour différente répartition pour un ensemble réparti en trois groupes. En abscisse et en ordonnée, elle représente le n_i de deux des ensembles, qui varient entre 0 (ensemble vide) et 1 (tous les éléments sont dans le groupe). Le troisième groupe étant linéairement dépendant des deux autres par construction, les deux premiers suffisent à représenter dans l'espace l'entropie. L'entropie de l'ensemble est maximale lorsque les trois ensembles contiennent le même nombre d'éléments (33 % dans chaque groupe) et vaut ici $log(3)$, soit environ 1,1. L'entropie est maximale lorsqu'il y a répartition égale des éléments dans les groupes. De l'autre côté, l'entropie est minimale lorsqu'un groupe représente la totalité et que les autres ne sont plus représentés.

La pertinence de l'entropie pour qualifier la sensibilité d'un binaire face aux attaques par détournement de flot d'exécution n'a pas été étudiée de manière approfondie. La figure 2.7 montre cependant les quelques résultats obtenus. Cette figure présente les différences par catégorie entre trois ensembles de binaires. Le premier est l'ensemble des binaires dont les sources sont en OCaml (environ 300 binaires), le deuxième est l'ensemble des binaires écrits en ADA du corpus (14

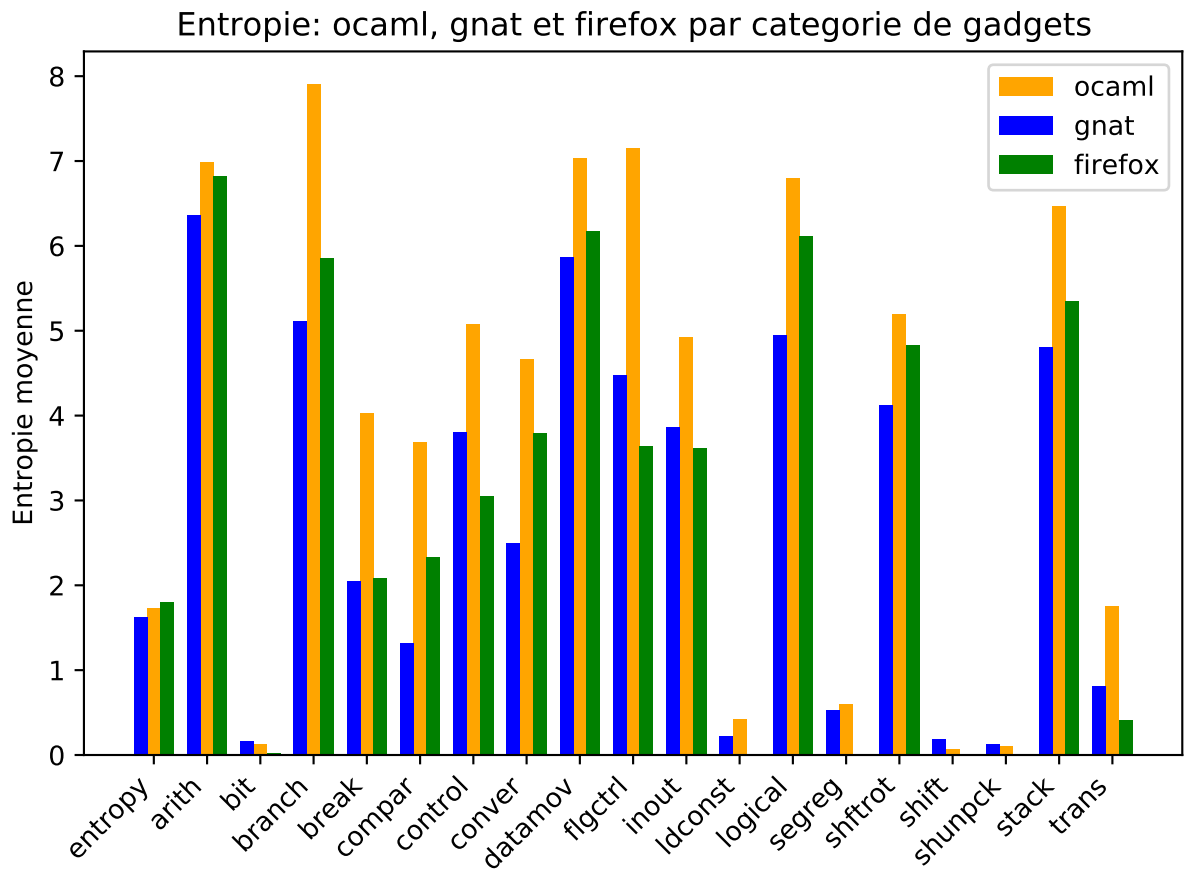


FIGURE 2.7 – Exemple de comparaison d'entropie par catégorie pour des binaires de différentes provenances

binaires) et le dernier est l'ensemble des versions du logiciel `Firefox`. L'idée derrière cette comparaison était dans un premier temps de voir si en moyenne l'utilisation de certains langages se démarque dans la diversité. Cette étude prend son sens en section 3.2, page 87 du chapitre 3. Les différences présentes dans cette figure ne sont pas suffisantes pour conclure sur l'utilisabilité de la métrique, mais cette figure montre suffisamment de différences pour justifier l'étude de l'influence des langages des sources sur la sensibilité des binaires. Le temps de calcul nécessaire pour étudier l'entropie et la faiblesse des résultats de la métrique sur l'ensemble initial ont écarté l'application de la métrique à tout le corpus.

2.3.2 Mesure de qualité

Pour permettre une classification rapide des gadgets utilisables par un attaquant et pour mesurer la difficulté d'utilisation d'un gadget Follner *et al.* [FBB16] ont mis en place une métrique de qualité d'un gadget. Chaque gadget a un score. Ce score prend en compte les effets de bord, les registres touchés et la catégorie du gadget. Ce score est calculé pour chaque gadget. Pour calculer leur score, ils commencent par caractériser les gadgets par leur première instruction. Dans leur modèle l'attaquant s'intéresse à la première instruction pour construire son attaque, la suite du gadget est du bruit jusqu'à l'instruction pivot. Un gadget a donc la forme suivante : `calc;(bruit;)* pivot`. Le score du gadget mesure sa difficulté d'utilisation. L'instruction `calc` ne contribue pas au score, car elle est l'instruction voulue, et il en va de même pour l'instruction pivot. Un gadget sans aucun bruit, `calc; pivot`, a un score de 0.

Ensuite ils mesurent à quel point le bruit est gênant. Pour cela, ils commencent par récupérer les registres dans lesquels écrit la première instruction, registres nommés r_d. Une unique instruction peut écrire dans plusieurs registres, notamment `xchg`, qui échange le contenu de deux registres. Les catégories des instructions bruyantes sont aussi considérées. Ils considèrent trois catégories : *data move*, *arithmetic* et *shift and rotate*. Les scores sont ajoutés au score du gadget pour chaque instruction bruyante selon le tableau 2.3.

Définition. *Soit un gadget* $X = (x_1, ..., x_n)$*, alors* $score(X) = \sum_{i=2}^{n-1} w(x_i)$*, où* $w(x_i)$ *est le poids de l'instruction donnée par le tableau 2.3*

Quelques gadgets et leurs scores sont donnés pour exemple :

— `add 0x5b,%al; ret`, score : 0
Pas d'instruction bruyante, le gadget est donc « parfait »

— `pop %rbx; pop %rbp; ret`, score : 2.5
Le registre destination est `%rbx`, une instruction bruyante modifie deux registres `%rbp` explicitement et `%rsp` implicitement.

Pour évaluer leur score, ils ont étudié ce qui impliquait l'activation de la protection MPX [2] mise en place par Intel. Cette protection, supportée par le processeur si le programme est compilé avec les bonnes options, permettait d'améliorer la vérification de validité des pointeurs d'accès mémoire. La protection est aujourd'hui

2. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html

Categorie	RSP	r_d	other
data move	2	1	0.5
arithmetic	2	1	0.5
shift and rotate	3	2	0.5

TABLE 2.3 – Contribution au score d'une instruction bruyante en fonction du registre touché en écriture

dépréciée, et n'est plus disponible dans leurs processeurs modernes. Pour mesurer la complexité de construction d'attaque induite par cette protection, la métrique utilisée est le dénombrement des gadgets ayant un score inférieur ou égal à 1.

Lorsque j'ai appliqué leur score à mon ensemble de binaires, j'ai aussi ajouté la moyenne de score des gadgets présents dans le binaire. Cet ajout est motivé par la volonté de pouvoir estimer la qualité générale des gadgets. Cela complète ainsi le dénombrement simple avec une mesure générale du bruit à corriger par les autres gadgets disponibles si besoin. Cette moyenne n'a cependant que peu de variation d'un binaire à un autre et il n'est pas possible d'en tirer des conclusions intéressantes.

La pénalisation du bruit induit par les instructions de type *shift and rotate* n'est justifié que par la difficulté à compenser leurs effets, ce qui est discutable. Bien que ce ne soit pas explicitement dit, les opérations de décalage peuvent avoir une perte d'information. Par exemple, un décalage à droite de deux bits induit la perte de la valeur des deux bits de poids faible. Le bruit d'une telle instruction doit donc être compensé par une sauvegarde au préalable de la valeur ainsi modifiée. Cependant, des effets similaires doivent être considérés pour les dépassements sur les additions ou multiplications par exemple. Les instructions *shift and rotate* sont donc considérées avec le même poids que les autres. Le calcul du bruit reste cependant sur leurs valeurs arbitraires de 2, 1 et 0.5 en fonction du registre touché en écriture. La pondération en fonction du type de registre touché est intéressante et n'est pas modifiée.

Pour les catégories considérées pour la sensibilité des binaires, j'ai aussi détaillé le contenu de certaines catégories. La liste des catégories qu'ils considèrent est :

— *arithmetic*
— *data move*
— *control flow*
— *logic*

— *RETs*
— *shift and rotate*
— *flag*
— *NOP*
— *floating Point*
— *misc*

Dans le cadre d'une attaque, certaines autres catégories peuvent avoir un intérêt. Par exemple, être capable de gérer des entrées et sorties ou faire de la conversion peut être intéressant. Sur des processeurs un peu plus vieux, les instructions gérant les registres de segmentation sont aussi utiles. Les catégories qui sont ajoutées par rapport à l'article original sont les suivantes, et correspondent aux catégories décrites en section 1.4.2 :

— *I/O*
— *conver*
— *segreg*
— *system*
— *break*

La catégorie *RETs* a été intégrée à la catégorie *control flow*. Les gadgets contenant une instruction de contrôle de flot d'exécution sont tout aussi délicats à utiliser et le type de l'instruction en soit n'importe que peu.

Notre outil nous permet de récupérer des gadgets utilisant des instructions très particulières (MMX, SIMD, etc.), ces gadgets sont néanmoins peu nombreux et difficiles à utiliser lors d'une attaque. Nous avons donc choisi de ne pas les intégrer à l'analyse de qualité. Moins de 1 % des gadgets analysés ont ainsi été mis de côté.

Le score mis en place dans [FBB16] est utile pour un attaquant qui cherche des gadgets dans un binaire. Faire un tri des gadgets par score permet de faire ressortir les gadgets facilement utilisables pour construire une chaîne. En revanche, cette métrique n'est pas pertinente pour déterminer la sensibilité d'un binaire. Un traitement statistique, comme le calcul de score moyen et variance de score des gadgets dans un binaire n'est pas suffisamment précis pour mettre en évidence un binaire plus ou moins sensible aux attaques par détournement de flot de contrôle. La variation de la métrique d'un binaire à un autre est trop faible pour cela. En effet, la variation observée d'un binaire à l'autre est de l'ordre de 5 %.

C'est pourquoi j'ai plutôt mis en place les calculs statistiques de score en fonction de la catégorie de la première instruction d'un gadget. Cela me permet de voir sur un binaire la prédominance de gadgets utilisables en fonction des

GAS	Intel
RET	RETN
LJMP	JMP
MOVSB	MOVS
FUCOMPI	FUCOMP
MOVABS	MOV

TABLE 2.4 – Traduction d'une instruction en assembleur GNU vers l'assembleur Intel

catégories. Cette distinction permet une comparaison plus fine que ce qui est proposé précédemment. Les variations observées d'un binaire à un autre sont suffisamment importantes pour servir de point de comparaison. Cette métrique est utilisée en section 3.2, pour comparer la sensibilité et la diversité des gadgets d'un langage de programmation à un autre.

2.3.3 Équivalence de gadgets

La classification syntaxique des gadgets a quelques limitations sur le dénombrement de gadgets uniques présents dans un binaire. Considérons les deux gadgets suivants : `pop %rbx; add %rax,0; ret` et `add %rax,0; pop %rbx; ret`. Ils sont syntaxiquement différents et donc classés comme tels dans notre analyse précédente. Ils font cependant la même opération, dans un ordre différent. Les instructions `add` et `pop` sont ici commutables, car elles utilisent des registres différents. Cette section présente une forme normale de gadget permettant d'identifier ces deux gadgets comme équivalents.

Deux instructions sont commutables si la première n'écrit pas dans les registres lus par la seconde et si la seconde n'écrit pas dans les registres lus par la première.

Définition. *Soit deux instructions i et i'.*
$i \leftrightarrow i'$ *ssi* $w(i) \cap r(i') = \varnothing \wedge w(i') \cap r(i) = \varnothing \wedge w(i') \cap w(i) = \varnothing$
où $w(i)$ (resp. i') est l'ensemble des registres potentiellement écrits par i (resp. i') et $r(i)$ (resp. i') l'ensemble des registres potentiellement lus par i (resp. i')

Les registres implicites sont aussi considérés. Par exemple l'instruction `pop %rbx` modifie le registre `%rsp`, puisqu'il est décrémenté de 8. L'instruction `add %rax,0` va potentiellement toucher les fanions du registre `EFLAGS`. Ce registre est donc toujours considéré comme modifié par cette instruction. Enfin les instructions critiques qui changent le flot d'exécution ou le contexte d'exécution sont

toujours définies comme non commutables. Par exemple les instructions `syscall`, `break` ou encore `sysexit` sont présentes dans cette catégorie. Pour déterminer les w et r de toutes les instructions de la famille x86, je me suis basé sur la référence des instructions disponible ici [3]. L'intégralité des registres touchés, accès mémoires et autres particularités des instructions de l'architecture y est détaillé.

L'assembleur utilisé dans le document est celui d'Intel contrairement à celui qui est utilisé par `ROPgadget`. Une traduction de certaines instructions de l'assembleur GNU (GAS) vers l'assembleur Intel a été réalisée. Par exemple l'instruction `RET` utilisée par `ROPgadget` n'est pas présente dans le document à disposition, qui utilise `RETN`. La correspondance étant directe avec le document, quelques instructions ont nécessité ces traductions supplémentaires. Le tableau 2.4 liste toutes les traductions que j'ai opérées. Certaines traductions avec changement d'opérandes entre les formalismes, par exemple implicite dans l'assembleur Intel et explicite pour GAS, ne sont pas précisées dans le tableau.

Ensuite une relation d'ordre entre les instructions a été définie. Cette relation va permettre de mettre en place une forme minimale d'un gadget en triant les instructions commutables du corps du gadget. Cette relation d'infériorité d'une instruction sur une autre est applicable sur deux instructions commutables. Soit deux instructions a et b telles que $a \leftrightarrow b$. On définit la relation d'ordre entre a et b en algorithme 1.

On définit la commutation adjacente de deux gadgets comme suit :

Définition. $X = (x_1, ..., x_n) \equiv Y = (y_1, ..., y_n)$ *ssi* $\exists i < n, x_i = y_{i+1}, x_{i+1} = y_i, x_i \leftrightarrow x_{i+1}, \forall j < n, j \neq i$ *et* $j \neq i+1 \Rightarrow x_j = y_j$

Cette définition n'est cependant pas transitive. Elle n'est donc pas suffisante pour lier des gadgets entre eux. Elle permet quand même d'avoir une première classe de similitude pour des gadgets. Cette première relation est utilisée pour mettre en place une relation d'équivalence ensuite.

Pour mettre en place cette relation d'équivalence transitive, on effectue une complétion transitive de l'équivalence précédemment définie. Pour cela, j'ai commencé par mettre en place une relation d'ordre sur l'ensemble des formes que peut prendre un gadget.

Définition. *L'ordre de deux gadgets X et Y est donné par l'ordre de leur première instruction distincte.*

3. http://ref.x86asm.net/

if *b fait au moins un accès mémoire et pas a* **then**
| $a \prec b$
else if *(a et b) ou (ni a ni b) n'ont d'accès mémoire* **then**
| **if** *b modifie au moins un registre et pas a* **then**
| | $a \prec b$
| **else**
| | Soit n_a^o et n_b^o le nombre d'opérandes de a et b;
| | **if** $n_a^o < n_b^o$ **then**
| | | $a \prec b$
| | **else**
| | | L'ordre de a et b est l'ordre lexicographique de leur premier opérande respectif distinct;
| | **end**
| **end**
else
| $b \prec a$
end

Algorithme 1 : Ordre entre deux instructions commutables a et b

L'ensemble des permutations que l'on peut atteindre en partant de X, en ne faisant que des permutations légales, peut être ordonné avec la relation d'ordre précédemment définie. La procédure de réduction d'un gadget est comme suit :

$X = (x_1, ..., x_n)$;
while $\exists i < n$ *tel que* $x_i \leftrightarrow x_{i+1}$ *et* $x_{i+1} < x_i$ **do**
| $X = (x_1, ..., x_{i+1}, x_i, ..., x_n)$;
end

Algorithme 2 : Réduction en forme minimale

La forme de X obtenue à la fin de l'exécution de l'algorithme est appelée forme minimale de X. En se basant sur cette réduction, on peut définir une relation d'équivalence généralisée de la manière suivante.

Définition. *Deux gadgets sont dits équivalents si et seulement si ils ont la même forme minimale.*

Cette nouvelle définition a l'avantage d'être transitive et permet de faire plus facilement des distinctions entre différents gadgets.

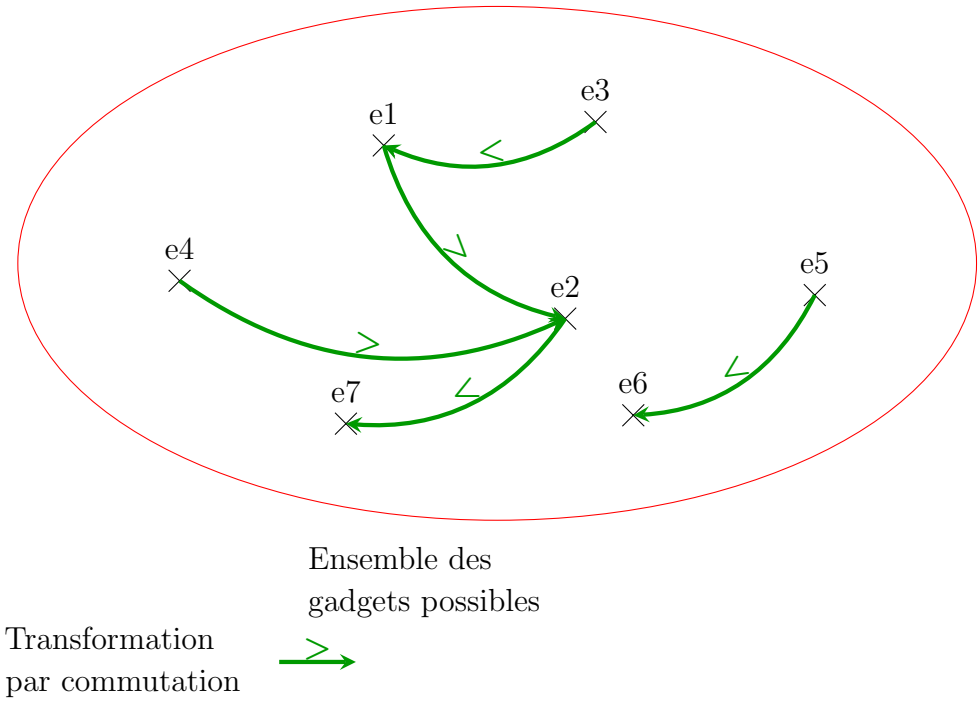


FIGURE 2.8 – Complétion transitive de la relation d'ordre des gadgets

Cette réduction et équivalence est illustrée en figure 2.8. Cette figure présente l'ensemble des permutations d'un gadget qui peuvent exister. Sur cette figure on peut voir la réduction du gadget `e3` vers `e7` en passant successivement par les gadgets `e1` et `e2`. Le gadget `e4` se réduit aussi en `e7` en passant par le gadget `e2`. Les gadgets `e1`, `e2`, `e3`, `e4` et `e7` ont donc la même forme minimale `e7`. Ils sont donc équivalents. Les gadgets `e5` et `e6` sont composés des mêmes instructions, mais ils n'existent pas de commutations possibles pour réduire `e6` vers un des autres gadgets. `e5` et `e6` ne sont donc pas équivalents aux autres gadgets écrits avec les mêmes instructions.

Réalisation

J'ai donc utilisé cette relation d'équivalence pour étudier de manière plus précise l'unicité des gadgets présents dans des binaires. Les résultats précédemment peuvent donc être consolidés en ne travaillant plus sur l'unicité syntaxique des gadgets, mais sur les gadgets équivalents.

Ce travail de consolidation n'a été fait que sur la famille d'instructions x86. Les

résultats présentés dans la section suivante ne concernent donc que ces architectures. La méthodologie utilisée ici n'est pas incompatible avec d'autres architectures. Il faut cependant classifier les instructions de toutes ces architectures ou utiliser un formalisme d'instruction indépendant de l'architecture. Les résultats sur l'architecture x86, qui représente environ 85 % du corpus, n'ont pas pu justifier le travail à fournir pour faire cette adaptation.

2.3.4 Étude d'unicité des gadgets

Que ce soit dans un même binaire, ou dans des binaires différents, on cherche ici à déterminer s'il est possible de trouver deux gadgets distincts qui ont la même forme normale. C'est-à-dire, deux gadgets différents qui à partir du même état du processeur et de la mémoire arrivent dans exactement le même état. L'idée est de pouvoir mieux caractériser l'unicité des gadgets autrement qu'en ne qualifiant que leur syntaxe.

Le premier résultat observé sur cet ensemble d'analyse est que la notion d'unicité des gadgets dans un binaire ne change pas en passant de l'écriture syntaxique de chaque gadget à leur forme minimale respective. Dans un même binaire, il y a une association unique entre un gadget syntaxique et sa forme minimale. On peut donc facilement conclure qu'un compilateur, avec des options de compilation fixes, produit toujours le même gadget pour effectuer une action précise. Ce résultat, bien qu'intéressant peut venir de notre échantillon. Bien que le corpus soit assez large, tous les compilateurs et langages ne sont pas analysés. Le langage ADA, par exemple, ne représente qu'une dizaine de binaires, ce qui n'est pas significatif statistiquement parlant. Et bien que que la recherche de gadgets inclut les gadgets décalés dans un binaire, il est intéressant de voir que même ainsi, on ne trouve pas deux gadgets syntaxiquement différents ayant la même forme minimale.

Dans un second temps, sur un sous-ensemble de binaires plus proches, les différentes versions de `tar` notamment, j'ai regardé si deux gadgets distincts présents dans deux binaires différents avaient la même forme minimale. Par exemple, il est intéressant de savoir si `gcc` et `clang` peuvent produire deux gadgets différents pour un comportement similaire. Sur cet ensemble de binaires, il n'existe aucun couple de gadgets distincts ayant la même forme minimale. Cet ensemble regroupe 183 494 gadgets en forme minimale, présents dans au moins deux binaires différents. Le nombre total de formes minimales, incluant celles présentes dans un seul binaire,

est de 311 016.

D'un point de vue d'un attaquant, ce résultat est cependant assez intéressant. Pour construire une chaîne, un attaquant va être sûr qu'il ne trouvera pas deux gadgets distincts ayant la même forme minimale. Il peut donc simplifier la recherche de gadget dans un binaire car tous les gadgets faisant la même opération ont la même séquence d'instructions.

2.4 Conclusion

La sensibilité d'un logiciel vis-à-vis des attaques par détournement de flot d'exécution peut être mesurée. Le chapitre a présenté le corpus de binaires utilisés pour la mise en place d'une étude extensive de la sensibilité au détournement de flot d'exécution de logiciels. Ce chapitre justifie les différents paramètres pris en compte lors de la sélection des systèmes étudiés, comprenant l'architecture matérielle, le type de distribution des binaires et l'objectif métier du système.

Ce chapitre a ensuite proposé plusieurs métriques pour étudier la sensibilité d'un binaire face au détournement de flot d'exécution. La première métrique est le dénombrement des gadgets comme ce qui est fait dans la littérature dans un premier temps [QPY18 ; Zha+13 ; Ona+10]. Cette métrique est ensuite rapportée à la taille de section exécutable dans le binaire pour définir la densité de gadget. Dans la continuation des métriques quantitatives, une notion d'entropie a été introduite pour mettre en place les fondements d'une mesure de diversité des gadgets dans un binaire. Pour améliorer ces métriques quantitatives, une notion de qualité des gadgets a été utilisée, en se basant sur les travaux de Follner et al. [FBB16]. Enfin, afin de valider l'approche d'analyse syntaxique des gadgets, le chapitre a fini sur l'étude de la réduction des gadgets dans une forme minimale.

Ce chapitre établit un ensemble de métriques pour mesurer la sensibilité de binaires face au détournement de flot d'exécution. Les résultats présentés dans ce chapitre ne permettent pas de déterminer la provenance des gadgets intéressant un attaquant, et comment améliorer la transformation des sources d'un logiciel en binaire plus difficilement attaquable. Le chapitre 3 utilise les métriques introduites ici pour étudier l'influence des moyens de production de binaires. Le chapitre 4 quant à lui utilise ces métriques pour mettre en place des méthodes de conception permettant d'influer sur la sensibilité des binaires associés aux logiciels à protéger.

Chapitre 3

Sensibilité d'un logiciel

Lors de la conception d'un logiciel, un développeur peut faire des choix variés à différents niveaux de la conception. Ces choix sont motivés par les objectifs à remplir par ce logiciel et les contraintes imposées. Ces contraintes peuvent être relatives aux performances ou à la sécurité, par exemple. Parfois, ces choix peuvent être dus à la compétence de l'équipe de développement dans une technologie qui sera choisie.

La question de la protection vis-à-vis des attaques par détournement de flot d'exécution arrive tardivement dans les étapes de conception et certains choix ont déjà été effectués. Un langage, un compilateur ou une plateforme matérielle sont sélectionnés, et ensuite une protection peut-être utilisée si elle est compatible avec ces choix.

Le chapitre précédent a défini un ensemble de métriques permettant de caractériser la sensibilité d'un binaire vis-à-vis des attaques par détournement de flot d'exécution. Dans ce chapitre, les métriques définies sont utilisées pour déterminer l'influence que le processus de création d'un logiciel peut avoir sur cette sensibilité. Le chapitre est composé de trois parties inégales en termes de contenu. La première concerne l'étude de compilateurs et des options qui leur sont associées. C'est l'étude la plus complète du chapitre. Le chapitre continue ensuite sur une étude, plus succincte, de l'influence du langage sur la sensibilité d'un binaire face au détournement du flot d'exécution. Enfin, ce chapitre termine par une étude assez limitée sur l'influence de l'architecture matérielle.

3.1 Le compilateur

3.1.1 Introduction

La génération du binaire d'un logiciel va dépendre du compilateur et du choix des options utilisées avec ce compilateur. Une option influe grandement sur le binaire produit, par exemple en enlevant du code qui n'est pas utilisé avec une des différentes optimisations disponibles. La sensibilité d'un logiciel au détournement de flot d'exécution dépend donc du processus de génération du binaire. Les sources d'un seul logiciel peuvent avoir des binaires des sensibilités variables en fonction de ces choix.

Au chapitre 1, nous avons vu que certaines protections suppriment autant que possible les gadgets non-alignés, et sont gênées par les actions tardives du compilateur. L'option `lto` (*Link Time Optimisation*) par exemple est souvent responsable de déplacement de code favorisant l'apparition de nouveaux gadgets. Pour supprimer du binaire le code inutilisé, une technique dite de *debloating* peut entrainer elle aussi une introduction de gadgets absents initialement dans le binaire comme montré par Brown [BP19].

Cette section montre l'influence d'un ensemble de compilateurs et d'options sur la sensibilité de binaires produits en étudiant le comportement du logiciel couramment utilisé `tar`. Le logiciel est suffisamment gros en termes de taille de section exécutable et fonctionnalités pour permettre une étude statistique approfondie de l'effet des compilateurs. Les différents binaires ont une taille variant de 200 ko à un peu plus de 800 ko.

Les métriques définies au chapitre précédent vont permettre de valider la protection apportée par le compilateur `gfree` en plus de montrer les différences entre deux autres compilateurs largement disponibles, `gcc` et `clang`. Le compilateur `gfree` est basé sur `clang`. Les options considérées sont des options courantes, comme celles d'optimisation, et des options orientées sécurité, comme par exemple les canaris de protection de pile d'exécution. Bien que les compilateurs n'aient pas exactement les mêmes algorithmes d'optimisation et implémentent certaines protections différemment, les ensembles d'options ont été choisis pour qu'un équivalent existe chez l'autre compilateur. L'option `-fstack-protector-all` est par exemple présente à la fois pour `gcc` et `clang`, et n'ont en commun que la généralisation de la protection à toutes les fonctions, pas la manière de déployer cette protection.

Chaque option n'a pas été utilisée seule, toutes les combinaisons d'options

possibles ont été utilisées pour générer les binaires. Certains ensembles d'options présentent des options incompatibles et n'ont donc pas de binaire associé. Par exemple, aucun binaire analysé n'a les deux options `-fstack-protector-all` et `-fstack-protector-strong`. Deux options d'optimisation différentes ne peuvent pas être utilisées conjointement non plus par exemple. Le nombre de binaires ainsi obtenu est de 407, sachant que certains groupes d'options ont produit des binaires identiques. Le nombre de binaires complètement distincts les uns des autres est de 149 binaires.

Cette section présente les résultats de sensibilité du logiciel vis-à-vis du détournement de flot d'exécution, en commençant par l'étude des métriques quantitatives, pour finir sur les métriques qualitatives.

3.1.2 Influence du compilateur sur le nombre de gadgets uniques

La première métrique utilisée dans cette étude est le nombre de gadgets uniques présents dans ces différents binaires. Bien que la métrique a été montrée comme insuffisante pour déterminer la sécurité d'un binaire vis-à-vis du détournement de flot d'exécution en section 1.8, page 35, les observations montrent déjà des différences notables entre les compilateurs.

La figure 3.1 présente les premiers résultats en utilisant cette métrique. D'un point de vue général, la taille des différents binaires, quels que soient les options utilisées, le compilateur `gcc` produit des binaires dont la taille reste du même ordre de grandeur. Le compilateur `clang` et son homologue `gfree` produisent quant à eux des binaires de grande taille, de plus de 600 ko. Ensuite en termes de gadgets uniques trouvés dans les binaires, plusieurs groupes bien distincts de binaires sont identifiables.

Les binaires produits par le compilateur `gcc` sont assez regroupés d'un ensemble d'options à un autre. Le compilateur `clang` quant à lui, a un comportement bien différent d'une option à une autre. Les amplitudes de quantités de gadgets uniques sont légèrement plus grandes qu'avec `gcc`, ayant environ 4000 gadgets uniques pour les binaires les moins denses contre 5000 pour `gcc`. Les deux compilateurs ont un maximum avec moins d'écart aux alentours de 8000. Mais surtout on observe que l'utilisation d'options ou combinaison d'options permet d'obtenir une réduction de gadgets comparable à ce qui est obtenu avec certains ensembles sur `gfree`. Ce

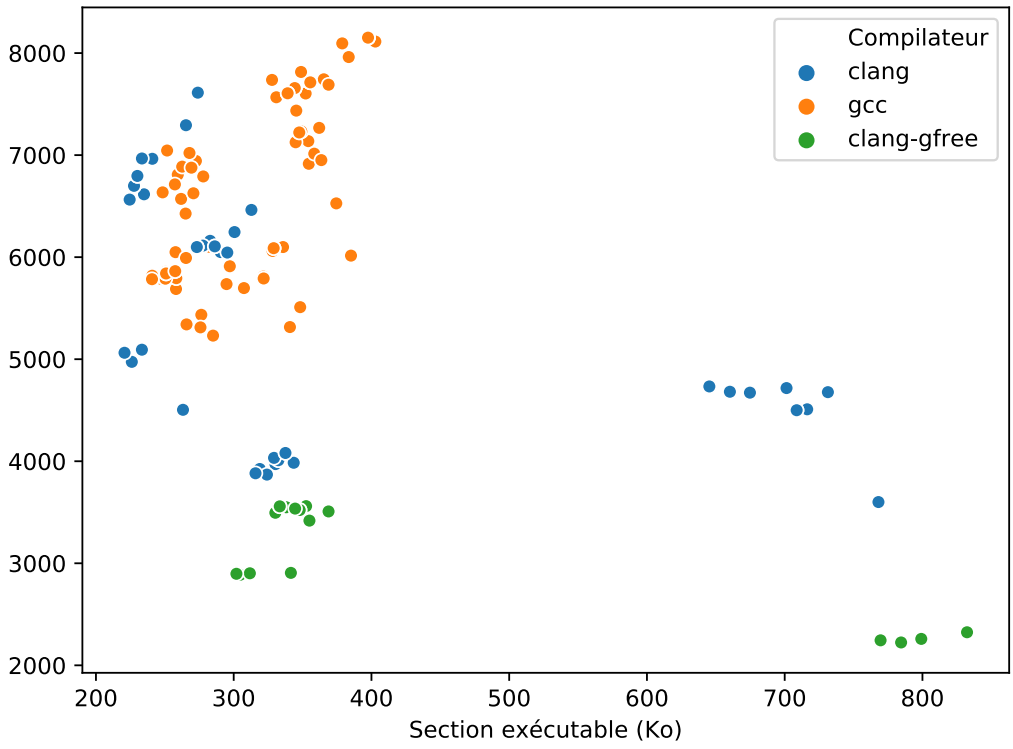


FIGURE 3.1 – Nombre de gadgets avec différents compilateurs et options pour le logiciel tar, version 1.30

dernier semble bien remplir sa fonction de réduction de gadgets. En effet, tous les binaires générés avec un autre compilateur ont plus de gadgets que n'importe lequel des binaires générés avec `gfree`.

Un regroupement des différents binaires produits par chacun des compilateurs permet d'identifier les options qui ont le plus d'influence sur le nombre de gadgets uniques. Pour mettre en place ce regroupement, un algorithme de *clustering* a été utilisé. Cet algorithme est DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*). La figure 3.2 présente ces résultats. Pour le compilateur `gfree`, les groupes sont assez distincts les uns des autres, avec peu de bruit. Le bruit correspond aux binaires que l'algorithme ne met dans aucun *cluster*. Pour `gcc`, les groupes sont moins visibles. Le calcul de la distance dans le plan a été modifié pour faire apparaître clairement les groupes de binaires en fonction de leur taille de section exécutable et nombre de gadgets uniques. Cette modification de calcul de distance a été aussi appliquée aux binaires produits par `clang` pour

affiner l'affectation des binaires aux groupes similaires.

Les binaires ayant une taille de section exécutable supérieure à 700 ko correspondent aux binaires n'ayant soit pas d'option d'optimisation utilisée soit forcée avec un `-O0`. Ce regroupement, noté **c1** pour le compilateur `clang` et **f1** pour `gfree` sur la figure, montre que, contrairement aux deux autres compilateurs, `gcc` effectue des opérations permettant la diminution de la taille du binaire même lorsqu'il est forcé de ne faire aucune optimisation. Les binaires ont une plus forte concentration de gadgets uniques par kilo-octet de section exécutable.

Pour le compilateur `gfree`, trois *clusters* sont bien différenciés. Les options d'optimisations sont bien séparées entre les différents groupes. Comme pour le

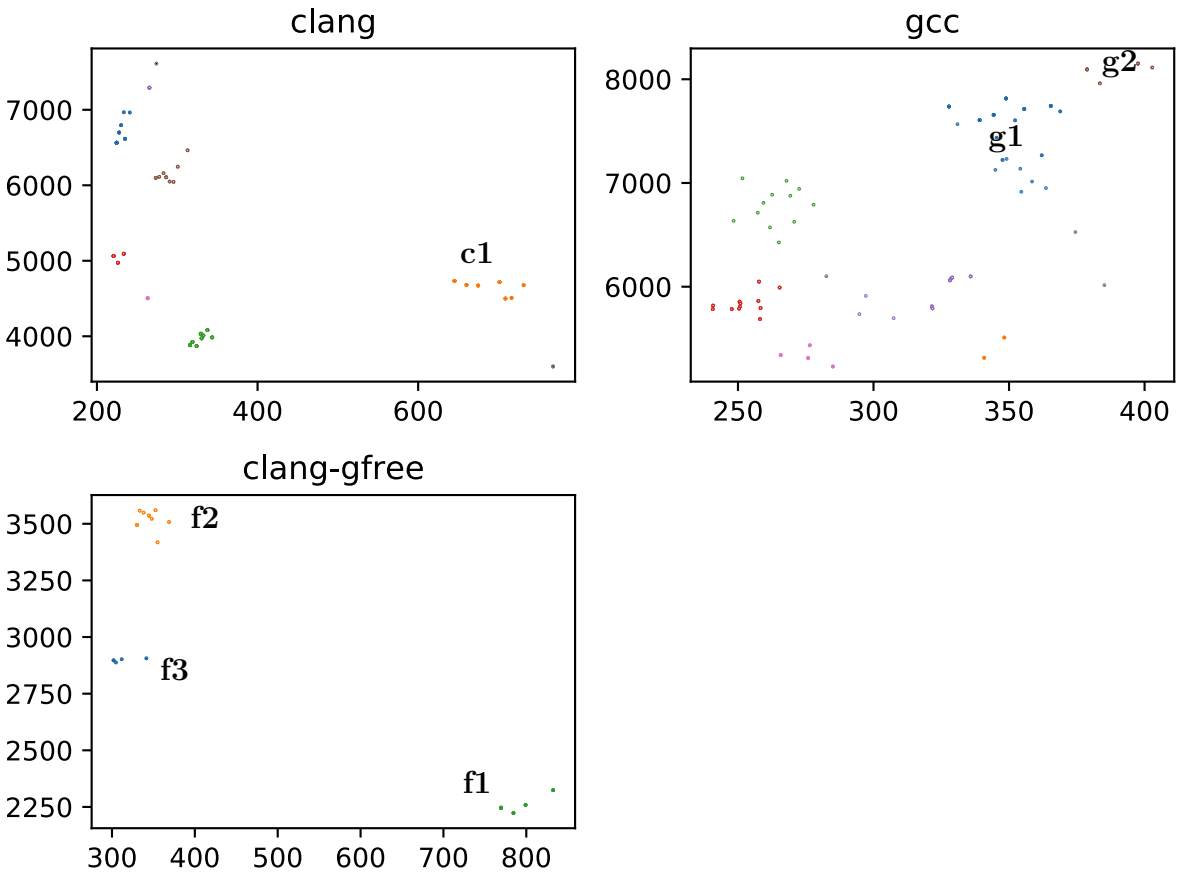


FIGURE 3.2 – Résultats avec application d'un algorithme DBSCAN pour les groupes de binaires similaires, séparés par compilateurs

compilateur non modifié `clang`, les binaires produits en forçant l'absence d'optimisation avec `-O0` ou sans options d'optimisations utilisées ont des tailles de section exécutable importante par rapport aux autres binaires et un nombre de gadgets uniques réduits. Le groupe est noté **f1** sur la figure 3.2. Les deux options `-O2` et `-O3` sont caractéristiques du groupe de binaires ayant le plus de gadgets uniques pour ce compilateur, visualisé par le groupe noté **f2**. Le compilateur est donc moins efficace sur des logiciels pour lesquels ces options sont nécessaires. Enfin l'option `-O1` produit des binaires ayant un nombre de gadgets uniques moyens vis-à-vis des autres options d'optimisations du compilateur tout en produisant des binaires un peu plus petits en moyenne. Ce groupe est noté **f3** sur la figure. Pour ce compilateur, aucune autre option n'est caractéristique d'un groupe de binaires. Il est à noter que, contrairement à `clang`, les optimisations tardives de compilation (*LTO, Link Time Optimisations*), ne sont pas disponibles.

La répartition des binaires produits par le compilateur `gcc` est plus concentrée qu'avec les autres compilateurs. Les différences sont plus subtiles sur les options qui influent sur le nombre de gadgets uniques présents. Chaque *cluster* est plus proche des autres que pour les autres compilateurs, moins de différences sont présentes d'un binaire à un autre en termes de nombre de gadgets uniques. Les *clusters* sont créés avec une distance seuil plus petite pour comparer l'influence des options du compilateur entre elles. Comparer l'influence en termes de quantité de gadgets d'une option similaire sur un autre compilateur n'est pas pertinent, car les distributions de binaires des compilateurs sont trop différentes. Le premier point identifié sur la répartition des binaires produits par `gcc` sont les deux *clusters* situés en haut à droite de la figure, notés **g1** et **g2**. Le premier, **g1**, est constitué de binaires n'ayant aucune option d'optimisation ou l'option `-O0`. Il s'agît du *cluster* qui contient les binaires ayant le plus grand nombre de gadgets uniques. Certains binaires sont compilés avec l'option `lto`, mais cela n'a aucune influence sur le binaire, `gcc` ignorant l'option. Tous les binaires de ces *clusters* ont une option en commun : `fstack-protector-all`. L'option est aussi utilisée pour des binaires présents dans d'autres *cluster*. Les autres options utilisées dans ce *cluster* n'ont soit aucun effet, soit un effet limité, le couple (`-O0`, `fstack-protector-all`) peut être considéré comme caractéristique de ce *cluster*.

Les cinq autres *cluster* de `gcc` ne sont pas tous aussi facilement distinguables, les options y sont plus variées. De la même manière pour le compilateur `clang`, l'absence d'optimisation va être caractéristique d'un *cluster*, tandis que les autres options ne le seront pas. Cette section nous montre que le dénombrement simple

des gadgets uniques dans un binaire permet de qualifier facilement quelques options. En effet, cette classification permet de montrer que la protection `gfree` fonctionne moins bien sur des programmes qui ont besoin de meilleures performances. La section suivante présente plus en détail l'influence des différentes options et compilateur. Le dénombrement de gadgets uniques ne suffisant pas, d'autres métriques peuvent être utilisées pour évaluer les changements de sensibilité apportés par les différents outils de compilation testés.

3.1.3 Catégorisation quantitative

L'analyse précédente concerne les gadgets uniques, indépendamment de leur utilité pour la construction d'une attaque par détournement de flot d'exécution. Pour chacune des catégories définies en section 1.4.2, l'influence des différents compilateurs varie. Cette section présente les variations pour les différents processus au regard du nombre total de gadgets et non plus uniques comme fait dans la section précédente. Le nombre de gadgets ainsi considérés double en moyenne par rapport à ce qui a été présenté dans la section précédente.

La figure 3.3 présente les résultats obtenus pour les trois compilateurs présentés précédemment et leurs options. Les catégories étudiées sont les catégories de gadgets permettant la construction de programmes d'attaque standard. Toutes les instructions de type MMX, FPU par exemple sont écartées, car elles ne sont pas pertinentes dans un cas d'attaque habituel, comme nous l'avons présenté en section 1.4.2, page 20.

Le premier élément d'étude porte sur le binaire qui contient le plus de gadgets. En effet, ce changement de dénombrement montre que le binaire en question n'est plus un binaire généré par `gcc` mais par `clang`, avec un nombre conséquent de gadgets en plus. Cette marge est due à une forte présence de gadgets de la catégorie `branch` dans ces binaires. Les binaires `gcc` ayant un fort nombre de gadgets sont principalement constitués de gadgets des catégories `arith` et `datamov`. Cette dernière catégorie montre aussi la plus grande variation quantitative des binaires produits par le compilateur `gcc` par rapport aux autres compilateurs.

Cette figure montre aussi que `gfree` diminue le nombre de gadgets disponibles dans chaque catégorie, à part pour les gadgets de la catégorie `branch`. Dans cette catégorie, le compilateur dédié à la protection contre le détournement de flot d'exécution produit plus de gadgets que la majorité des autres binaires. Les réductions dans les autres catégories sont cependant significatives, donnant une

bonne confiance d'un point de vue quantitatif sur la sécurité apportée par la solution. De plus, les gadgets contenant des instructions de type `branch` sont un peu plus compliqués à utiliser, puisque des conditions sur les registres sont à vérifier pour éviter que l'exécution de la chaîne soit interrompue par un saut non voulu.

Pour déterminer l'influence de chaque option, pour chacune des catégories, chaque binaire a été comparé à tous les binaires qui diffèrent à une option de compilation près. La mesure de l'influence d'une option de compilation d'un compilateur repose sur l'ensemble des binaires utilisant l'option. À chacun de ces binaires est associé le binaire ayant exactement les mêmes options, moins l'option dont on mesure l'influence. Par exemple, pour mesurer l'influence de l'option `-fstack-check` du compilateur `gcc`, le binaire ayant utilisé les options `-fstack-check` et `-O1` est associé au binaire n'ayant utilisé que l'option `-O1`. Les différences sur les métriques disponibles sont calculées sur l'ensemble des associations. Les tableaux 3.1 présentent les différences quantitatives en termes de gadgets uniques utilisables pour des attaques par détournement de flot d'exécution. Les dénombrements présentés sont la moyenne des différences trouvées entre deux binaires associés.

Certaines options n'ont parfois aucune influence lors de la génération du binaire. En effet, le compilateur peut ignorer des options, car elles sont incompatibles ou n'ont pas d'implémentation permettant l'intégration des deux options. Pour éviter d'avoir un biais avec une duplication des binaires ayant des options différentes et produisant exactement le même binaire, le corpus a été épuré des doublons. L'identification des doublons repose sur une somme de contrôle du binaire, l'algorithme SHA-256. Les options retenues pour le binaire qui représente le doublon est l'intersection de l'ensemble des options des binaires identiques. Par exemple si les binaires avec les options `-O1,-flto,-fstrict-vtable-pointers` et `-O1,-flto` sont identiques, l'ensemble d'options `-O1,-flto` est celui sélectionné pour représenter le binaire dans les analyses des options de compilation.

Plusieurs points sont intéressants à voir dans ce tableau 3.1. Pour le compilateur `gfree`, les résultats présents dans la troisième colonne, contenant l'ajout ou diminution du nombre de gadgets, ne sont pas étonnants. Les *clusters* initiaux vus en figure 3.2 sont bien séparés en fonction des options de compilation utilisées, et cette méthode d'étude d'influence confirme que ce sont ces options qui ont de l'importance pour protéger un binaire contre le détournement de flot d'exécution. Ce tableau montre aussi que les augmentations dans les catégories `branch`, `arith` et `stack` sont biens dues elles aussi aux options d'optimisations.

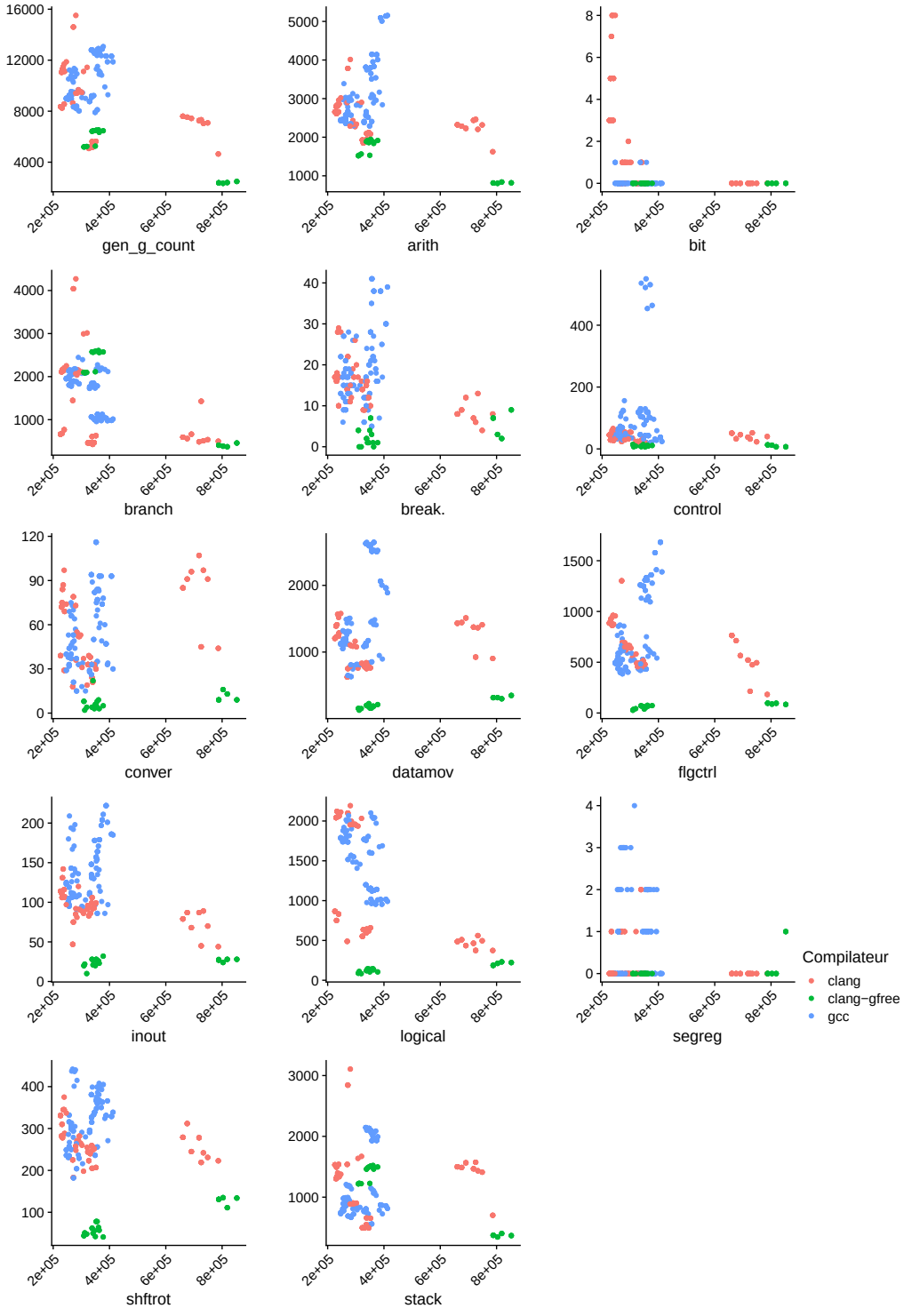


FIGURE 3.3 – Nombre de gadgets avec différents compilateurs et options pour le logiciel tar, version 1.30, dans les catégories de gadgets

Option	Diminution	Augmentation	Gadgets	arith	bit	branch	break	control	conver	datamov	flgctrl	inout	logical	segreg	shftrot	stack
-O0	1(moy : -3.0)	8(moy :0.0)	-0.3(0.9)	-0.2(0.6)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	-0.1(0.3)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)
-O1	0	9(moy : 2821.9)	2821.9(33.7)	717.3(10.5)	0.0(0.0)	1693.7(24.0)	-3.2(1.0)	0.1(2.8)	-6.1(5.2)	-170.3(16.4)	-56.3(9.3)	-8.7(5.5)	-113.8(19.2)	-0.2(0.4)	-82.1(10.6)	851.6(22.5)
-O2	0	9(moy : 4023.1)	4023.1(99.5)	1058.0(36.1)	0.0(0.0)	2165.4(38.3)	-4.3(2.9)	-0.6(4.8)	-3.9(5.6)	-118.9(30.6)	-23.8(5.2)	-1.8(2.2)	-77.1(11.6)	-0.2(0.4)	-71.2(10.5)	1101.4(18.6)
-O3	0	9(moy : 4094.9)	4094.9(63.8)	1094.3(30.8)	0.0(0.0)	2186.6(42.2)	-2.0(3.2)	-0.7(4.8)	-4.9(1.7)	-141.9(19.4)	-28.8(13.0)	1.1(2.1)	-82.6(20.0)	-0.2(0.4)	-61.4(17.2)	1135.3(12.2)
-fno-stack-protector	1(moy : -3.0)	8(moy :0.0)	-0.3(0.9)	-0.2(0.6)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	-0.1(0.3)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)
-fstack-protector	5(moy : -36.6)	4(moy :27.0)	-8.3(33.6)	-3.9(26.6)	0.0(0.0)	-13.1(7.8)	-3.3(1.2)	-1.4(3.2)	5.7(8.1)	-6.7(11.1)	-0.3(7.4)	-2.7(3.1)	22.6(2.2)	0.0(0.0)	0.2(6.9)	-5.3(19.3)
-fstack-protector-all	4(moy : -62.0)	5(moy :89.2)	22.0(78.7)	-13.0(24.0)	0.0(0.0)	12.2(30.8)	-1.1(3.0)	-2.0(4.2)	-0.7(0.8)	22.1(24.7)	2.4(13.0)	0.0(3.1)	13.7(19.1)	0.3(0.5)	-8.8(15.4)	-3.2(7.5)
-fstack-protector-strong	0	9(moy : 33.2)	33.2(12.5)	-1.4(40.2)	0.0(0.0)	-7.4(20.2)	-4.1(1.8)	-0.7(5.5)	1.1(3.4)	7.4(21.1)	7.0(11.7)	-2.7(4.1)	23.3(17.9)	0.0(0.0)	-10.2(9.2)	20.9(12.8)
-fstrict-vtable-pointers	0	24(moy : 0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)

(a) gfree

Option	Diminution	Augmentation	Gadgets	arith	bit	branch	break	control	conver	datamov	flgctrl	inout	logical	segreg	shftrot	stack
-O0	4(moy : -975.5)	0	-975.5(247.4)	-380.8(53.6)	-2.2(1.3)	-79.0(38.8)	-5.5(4.4)	-2.2(9.8)	39.0(19.4)	252.8(34.0)	-455.0(378.4)	-26.0(17.2)	-283.2(114.7)	-0.2(0.4)	-39.0(40.8)	6.0(26.7)
-O1	0	8(moy : 4854.5)	4854.5(2523.0)	707.8(688.7)	4.1(3.2)	1972.6(765.6)	10.6(8.2)	2.6(15.5)	12.2(27.5)	117.4(99.1)	120.2(354.8)	23.6(20.5)	1492.2(203.3)	0.0(0.5)	18.9(70.4)	372.1(901.2)
-O2	4(moy : -3197.8)	4(moy :3396.2)	99.2(3531.0)	-216.6(666.5)	-0.8(1.8)	705.6(1150.4)	1.8(6.7)	-10.6(17.5)	-18.6(19.9)	-267.2(175.7)	-171.2(390.9)	8.8(33.5)	683.9(813.3)	0.2(0.4)	-34.2(50.9)	-581.6(611.6)
-O3	4(moy : -3065.2)	4(moy :3484.2)	209.5(3543.6)	-181.0(639.9)	-0.5(1.8)	715.1(1159.6)	5.4(6.9)	-7.0(16.7)	-28.5(26.9)	-250.1(187.5)	-159.0(402.8)	8.9(26.5)	696.4(814.7)	0.0(0.5)	-19.5(46.4)	-570.6(617.0)
-ftto	12(moy : -3295.3)	4(moy :1939.0)	-1986.8(2893.3)	-136.2(497.4)	-0.2(1.8)	-840.3(1041.9)	-2.6(7.2)	-11.4(15.1)	-18.1(25.5)	-201.1(104.7)	75.6(345.1)	6.4(19.7)	-615.4(740.5)	0.1(0.7)	-13.7(52.5)	-229.9(423.0)
-fstack-protector	6(moy : -102.7)	3(moy :163.7)	-13.9(149.0)	-28.0(97.8)	0.2(0.8)	7.4(35.5)	1.6(3.7)	6.1(10.7)	3.4(15.2)	-0.1(32.7)	-18.0(28.6)	-4.4(14.3)	32.1(59.5)	0.0(0.5)	-3.7(22.7)	-10.6(22.1)
-fstack-protector-all	2(moy : -1450.5)	7(moy :1751.4)	1039.9(1914.1)	357.3(550.3)	-2.0(2.5)	872.8(707.0)	-1.6(5.3)	-4.7(9.4)	-27.7(21.7)	-415.0(231.5)	-115.3(248.2)	-24.4(26.0)	-19.2(149.0)	0.3(0.8)	-66.1(26.4)	485.4(743.3)
-fstack-protector-strong	4(moy : -107.0)	5(moy :208.2)	68.1(182.7)	39.6(111.4)	-0.1(0.3)	57.9(43.0)	-0.3(4.5)	0.6(14.7)	-7.6(10.8)	-10.2(53.3)	-8.8(78.4)	-6.6(19.6)	12.6(39.0)	-0.1(0.3)	-9.8(19.8)	1.0(36.2)

(b) clang

Option	Diminution	Augmentation	Gadgets	arith	bit	branch	break	control	conver	datamov	flgctrl	inout	logical	segreg	shftrot	stack
-O1	15(moy : -2934.6)	0	-2934.6(847.3)	-1422.3(639.9)	0.1(0.2)	1087.8(170.8)	-10.5(9.8)	-148.4(218.8)	-32.9(24.9)	-1338.1(129.4)	-690.2(272.3)	-39.7(47.4)	670.8(153.0)	0.7(1.2)	-112.7(20.6)	-899.3(488.2)
-O2	15(moy : -2847.3)	0	-2847.3(1034.5)	-1497.2(715.9)	0.0(0.0)	937.1(166.9)	-10.3(10.1)	-107.1(194.8)	-33.8(18.0)	-1237.1(161.1)	-792.7(218.6)	-63.1(33.5)	720.9(213.0)	1.3(1.4)	-25.3(57.8)	-740.1(412.6)
-O3	15(moy : -2880.9)	0	-2880.9(1105.5)	-1449.5(696.6)	0.1(0.3)	941.6(230.1)	-12.3(11.0)	-84.3(219.4)	-31.9(30.4)	-1245.7(190.9)	-786.4(209.2)	-64.8(36.9)	769.6(197.6)	0.7(0.9)	-33.1(40.9)	-885.0(441.6)
-ftto	27(moy : -1525.9)	4(moy :361.2)	-1282.4(897.0)	-367.5(318.9)	0.0(0.3)	-205.8(168.8)	-1.6(9.5)	-88.9(170.4)	-10.9(24.3)	-164.3(150.2)	-118.8(153.1)	-20.4(33.0)	-81.1(162.8)	-1.7(1.1)	-59.5(38.1)	-161.8(111.6)
-fstack-check	16(moy : -280.1)	15(moy :167.3)	-63.6(306.5)	-26.4(155.1)	-0.0(0.3)	-4.4(46.7)	-0.7(9.0)	0.1(14.3)	2.8(16.3)	-3.7(54.3)	-8.7(61.3)	0.8(23.5)	-17.9(46.2)	0.1(0.6)	3.1(20.6)	-8.7(37.7)
-fstack-protector	4(moy : -128.2)	11(moy :174.7)	93.9(208.9)	55.9(110.8)	-0.1(0.4)	20.3(33.6)	1.8(8.5)	-0.6(13.2)	-2.0(11.3)	9.9(31.6)	13.1(61.6)	8.2(18.3)	-9.3(29.6)	0.1(0.5)	-10.4(18.0)	6.9(30.8)
-fstack-protector-all	15(moy : -1022.0)	0	-1022.0(516.2)	291.5(580.8)	-0.1(0.3)	70.9(120.9)	2.8(9.3)	-59.0(125.9)	-23.7(19.9)	-497.7(136.5)	5.7(171.2)	-16.1(38.8)	-280.3(159.7)	0.0(0.6)	-77.5(36.6)	-438.6(416.0)
-fstack-protector-strong	4(moy : -106.2)	11(moy :312.1)	200.5(248.1)	152.5(153.3)	-0.1(0.3)	74.9(33.6)	1.7(10.4)	-16.5(22.3)	-3.8(17.7)	-18.5(54.2)	60.3(47.4)	11.7(28.7)	-20.3(48.7)	0.0(0.6)	-6.7(15.7)	-34.7(80.3)

(c) gcc

TABLE 3.1 – Différence de gadgets apportée par chacune des options pour chacun des compilateurs. Ces options ont été testées sur le logiciel `tar` en version 1.30

Les options qui font varier la quantité de gadgets par catégorie sont plus diversifiées pour le compilateur `gcc`. Tout d'abord, l'option `-O0` n'est pas présente, car toute compilation avec `gcc` sans aucune optimisation précisée utilise l'option `-O0` en l'absence d'autres options. Ensuite, malgré un écart type assez important, les trois options d'optimisation classiques diminuent en moyenne le nombre de gadgets présents dans le binaire. Cette diminution de gadgets est principalement observable sur les catégories `arith`, `datamov` et `stack`. Les diminutions observées dans ces trois catégories ne dépendent que très peu du choix de l'optimisation. En effet, quelle que soit la catégorie de gadgets choisie, l'influence de l'optimisation est similaire. En contrepartie, l'utilisation des options entraine une augmentation des gadgets disponibles dans les catégories `branch` et `logical`. L'utilisation des options d'optimisation permet donc à un développeur de réduire la surface utilisable pour faire des calculs et déplacer des données, au détriment d'une surface accrue pour contrôler le flot de l'exécution. Ensuite l'utilisation de `LTO`, a des résultats plus variables. En termes de gadgets totaux, l'utilisation de l'option dans un cadre de protection contre les attaques par détournement de flot d'exécution va donc aussi dépendre des autres options utilisées conjointement lors de la génération du binaire. Une telle attaque ayant besoin d'une corruption mémoire pour être initiée, l'utilisation des canaris de protection de pile semble être une bonne idée. Leur utilisation généralisée sur l'ensemble du programme, avec l'option `-fstack-protector-all`, donne des bons résultats sur le nombre de gadgets présents dans le binaire final, en ne faisant augmenter que le nombre de gadgets de la catégorie `arith` de manière significative. Laisser le compilateur choisir quand mettre les canaris, avec l'option `-fstack-protector`, augmente légèrement la quantité de gadgets présents, principalement du côté des gadgets `arith` là aussi. Enfin, la dernière option de protection de pile considérée, `-fstack-protector-strong`, a des effets similaires.

Pour le compilateur `clang`, les choses sont bien différentes. En effet, la plupart des options de compilation ont des variations ayant de très forts écarts types. Même dans les cas où en moyenne l'option entraine une augmentation ou une diminution du nombre de gadgets, les variations sont trop importantes pour donner un indice valable sur la sensibilité potentielle apportée par une telle option. Les seules options du compilateur qui n'ont un effet que dans une seule direction dans le cas général sont les options `-O0` et `-O1`. Ces deux options provoquent respectivement sur notre corpus une diminution du nombre de gadgets pour la première et une augmentation significative pour la seconde. Les autres options d'optimisation comptent autant de cas avec une augmentation qu'une diminution, et dans les deux cas, la variation a une forte moyenne. Toutes les autres options ont un effet similaire sur le nombre

Options	Taille	Gadgets totaux	arith	bit	branch	break	control	conver	datamov	flgctrl	inout	logical	segreg	shftrot	stack
'-fstack-protector', '-O1'	31	-55	-48	-100	-5	-100	-89	-98	-92	-96	-79	-95	N/A	-86	-11
'-fstack-protector'	10	-67	-63	N/A	-25	-77	-77	-84	-77	-81	-73	-62	N/A	-44	-76
'-fstack-protector-strong', '-O1'	29	-56	-48	-100	-7	-100	-73	-95	-90	-96	-90	-96	N/A	-86	-11
'-O2', '-fstack-protector-all'	18	-43	-36	N/A	-15	-100	-74	-92	-75	-88	-73	-93	N/A	-71	-11
'-fstack-protector-strong'	9	-66	-64	N/A	-31	-50	-70	-86	-79	-81	-60	-53	N/A	-52	-71
'-O1'	31	-55	-46	-100	-2	-86	-77	-91	-90	-97	-86	-96	N/A	-87	-13
'-fstack-protector-strong', '-O2'	20	-33	-24	-100	20	-95	-71	-94	-79	-90	-78	-93	N/A	-79	65
'-fstack-protector-all'	8	-46	-50	N/A	-8	12	-82	-80	-61	-54	-36	-41	inf	-40	-48
'-fstack-protector-strong', '-O3'	19	-31	-19	-100	24	-95	-78	-71	-82	-88	-71	-93	N/A	-75	69
'-fstack-protector', '-O3'	20	-32	-17	-100	23	-88	-83	-85	-86	-91	-71	-93	N/A	-71	69
'-O3'	20	-32	-17	-100	23	-59	-78	-90	-86	-92	-69	-94	N/A	-72	70
'-O1', '-fstack-protector-all'	25	-66	-62	N/A	-51	-73	-84	-83	-79	-94	-78	-95	N/A	-83	-61
'-O2'	21	-32	-18	-100	24	-82	-72	-95	-82	-89	-67	-94	-100	-76	65
'-fstack-protector', '-O2'	20	-31	-17	-100	26	-92	-78	-60	-83	-89	-74	-93	N/A	-82	67

TABLE 3.2 – Apports ou retraits induits par le compilateur `gfree`, à options constantes, par rapport au compilateur `gfree`, en pourcentage. Une valeur négative signifie une diminution de gadgets dans le binaire produit par `gfree`

de gadgets génériques trouvés dans les binaires. Pour ce compilateur, l'utilisation de chaque option doit donc être considérée au cas par cas en fonction de l'ensemble des options considérées. À l'exception des options `-O1` et `-O0` qui, relativement aux autres, sont plus stables en moyenne. Cela n'est pas le cas pour toutes les catégories, comme par exemple `stack`, sur laquelle les variations sont aussi significatives que pour les autres options. Il est d'autant plus remarquable que le compilateur se comporte d'une telle manière alors que le compilateur dérivé `gfree` montre des variations très faibles pour chacune des options partagées.

Pour identifier les différences de comportement entre le compilateur `clang` et son dérivé `gfree`, les binaires produits avec les mêmes options sont comparés dans le tableau 3.2. Les développeurs de la solution montrent une réduction importante du nombre de gadgets par leur compilateur en règle générale. Les résultats obtenus montrent que sur le sous-ensemble de gadgets considérés pour les attaques, cela est aussi toujours vérifié. La diminution minimale observée est de 31 % et elle monte jusqu'à 67 %. Cela se fait toujours au prix d'une section exécutable de taille plus importante, puisque la variation est entre +8 % et +31 %. La solution est efficace pour supprimer des gadgets de certaines catégories, par exemple `datamov` ou `logical` pour lesquelles la diminution peut aller jusqu'à 92 % et 95 % de gadgets supprimés. Sur d'autres catégories, notamment les deux plus importantes, `stack` et `branch`, les apports de la solution sont plus variables. Ces catégories présentent des augmentations parfois importantes de gadgets, des diminutions dans d'autres. Par exemple dans la catégorie `stack`, pour le binaire généré avec uniquement l'option `-fstack-protector-strong`, la diminution est de 71 %, alors que le binaire avec en plus l'option `-O2` présente une augmentation de 65 % de gadgets. Les augmentations de gadgets dans ces catégories semblent corrélées à l'utilisation des options de

compilation `-O2` et `-O3`. Ces résultats permettent de conclure dans un premier temps que l'utilisation d'une telle solution dans des cas nécessitant des optimisations fortes du compilateur donne moins de gadgets, qui permettent moins de calculs, mais plus faciles utiliser pour construire une chaine.

3.1.4 Qualité des gadgets par compilateur

Les résultats sur la quantité de gadgets de chaque catégorie apportent des informations concluantes sur la surface utilisable par un attaquant. Cette section montre les résultats obtenus cette fois en s'intéressant à l'utilisabilité des gadgets. Cela est fait en utilisant la métrique définie au chapitre précédent en section 2.3.2. Cette métrique permet d'avoir un score donnant la difficulté d'utilisation des gadgets présents dans un binaire. Il est à noter que cette métrique, contrairement

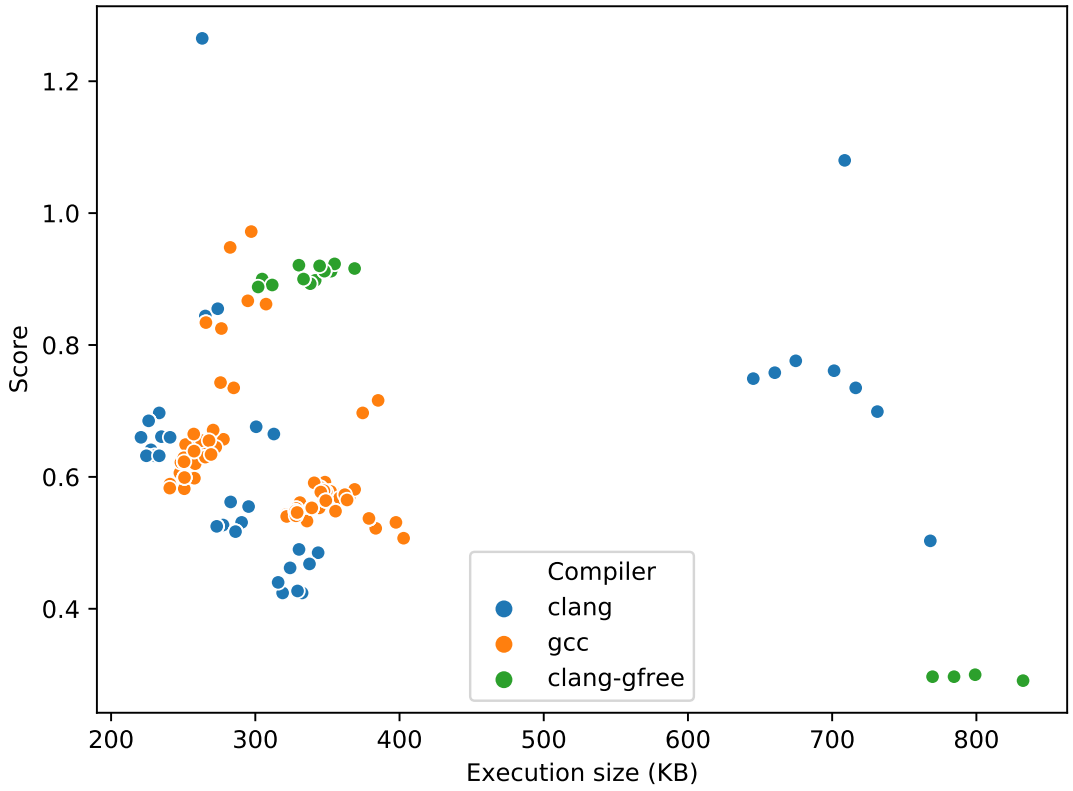


FIGURE 3.4 – Score moyen des gadgets avec différents compilateurs et options pour le logiciel tar, version 1.30

à ce qui a été fait dans le chapitre précédent, change légèrement la notion de catégorie de gadgets. En effet, notre analyse précédente considère l'intégralité des instructions précédent le saut pour déterminer l'ensemble des catégories auxquelles le gadget appartient, alors que pour l'utilisabilité, seule la catégorie de la première instruction du gadget est considérée. La section 2.3.2 de description de la métrique explique pourquoi ce choix a été fait.

La métrique précédemment définie donne un score à chaque gadget. Le gadget le plus facile à utiliser est un gadget ayant un score nul. Plus le score du gadget est important, plus il est difficile à utiliser. Cette difficulté est due par exemple à des effets de bords à prendre en compte, et des instructions parasites qu'il faut éviter d'activer. On peut penser par exemple à un saut conditionnel au milieu du gadget.

Pour comparer les différents compilateurs utilisés sur `tar`, dans un premier temps, le score moyen des gadgets de chaque binaire a été calculé. La figure 3.4 présente les résultats associés. Plusieurs points ressortent de cette figure. Tout d'abord, le compilateur `gfree`, qui protège donc les binaires contre du détournement de flot d'exécution, présente deux groupes bien différents en termes de score. Ces deux groupes sont caractérisés par l'absence totale d'optimisation d'un côté (`-O0` ou pas d'option), et la présence d'options d'optimisation de l'autre côté. Le score moyen des gadgets sur ces binaires passe d'environ 0,2 pour les binaires sans optimisation à environ 0,9. Les binaires ayant été compilés avec l'option `-O1` ont moins de gadgets que ceux utilisant les options d'optimisations supérieures, et ont un score moyen de gadget similaire. L'option `-O1` produit donc, selon ces métriques, des binaires plus difficiles à exploiter que les binaires utilisant les options `-O2` et `-O3`. Pour l'option `-O0`, il est plus difficile de conclure. En effet, les gadgets sont plus facilement utilisables, dus à leur faible score, mais il a été montré en section précédente que ces binaires présentent un nombre bien inférieur de gadgets. Les deux autres compilateurs ont des comportements plus variés. Le compilateur `gcc` va avoir un ensemble de binaires assez regroupé. De même, à quelques exceptions près pour le compilateur `clang`, la variation moyenne de score est assez faible. Pour ces compilateurs, l'évaluation des critères qui entrent en compte dans la position d'un binaire dans un groupe ne donne pas de résultat clair. On ne trouve pas d'option commune à certains groupes, comme cela est vérifié en terme quantitatif.

Cependant, comme cela a été relevé lors de la présentation de la métrique au chapitre précédent en section 2.3.2, le résultat d'un score moyen peu précis était attendu. En effet, certaines catégories étant sous représentée dans les binaires, certains points sont cachés par d'autres catégories plus larges. La section précédente

Option	average	bit	inout	datamov	arith	branch	shftrot	conver	break	logical	flgctrl	segreg	stack	control
-O0	0.0(0.1)	-1.3(0.8)	0.2(0.3)	0.3(0.1)	0.3(0.3)	-0.1(0.1)	0.0(0.1)	1.1(0.2)	-0.1(0.2)	-0.1(0.0)	-0.0(0.9)	-0.4(0.6)	-0.0(0.0)	-0.0(0.3)
-O1	-0.1(0.2)	0.8(0.9)	-0.1(0.2)	-0.2(0.2)	0.1(0.2)	-0.1(0.3)	-0.0(0.1)	-0.5(0.6)	0.4(0.3)	0.0(0.2)	-0.7(0.7)	0.1(0.9)	0.1(0.0)	-0.0(0.1)
-O2	-0.2(0.2)	-0.1(1.4)	-0.3(0.2)	-0.3(0.2)	-0.1(0.1)	-0.3(0.4)	-0.1(0.1)	-0.5(0.5)	0.3(0.3)	-0.1(0.2)	-0.8(0.8)	0.6(1.1)	-0.1(0.1)	-0.1(0.2)
-O3	-0.2(0.2)	0.1(1.3)	-0.3(0.2)	-0.3(0.2)	-0.1(0.1)	-0.3(0.4)	-0.1(0.1)	-0.6(0.5)	0.3(0.2)	-0.1(0.2)	-0.9(0.8)	0.0(0.8)	-0.1(0.1)	-0.2(0.1)
-flto	-0.0(0.2)	-0.0(1.2)	-0.1(0.2)	-0.1(0.1)	-0.0(0.1)	0.0(0.3)	-0.0(0.1)	-0.3(0.5)	-0.1(0.2)	0.0(0.1)	-0.0(0.7)	0.3(1.4)	-0.0(0.1)	-0.0(0.1)
-fstack-protector	0.0(0.0)	0.2(0.5)	0.0(0.1)	-0.0(0.0)	0.0(0.0)	0.0(0.0)	-0.0(0.0)	0.0(0.1)	-0.1(0.3)	0.0(0.0)	-0.0(0.1)	0.0(0.9)	0.0(0.0)	-0.0(0.3)
-fstack-protector-all	0.2(0.2)	-0.7(0.8)	-0.1(0.2)	-0.2(0.2)	0.1(0.2)	0.5(0.4)	-0.1(0.1)	-0.1(0.2)	-0.4(0.2)	0.1(0.2)	0.2(0.8)	0.6(1.7)	-0.1(0.1)	-0.2(0.2)
-fstack-protector-strong	0.0(0.0)	0.0(0.0)	-0.0(0.1)	-0.0(0.0)	0.0(0.0)	0.1(0.1)	-0.0(0.0)	0.0(0.1)	-0.1(0.2)	0.1(0.1)	-0.0(0.2)	-0.2(0.6)	0.0(0.0)	-0.1(0.1)

TABLE 3.3 – Influence de chacune des options utilisées pour le compilateur `clang`, en termes de score pour chacune des catégories, sur le logiciel `tar` en version 1.30

montre aussi cette représentation prépondérante de certaines catégories comme `arith` par rapport à `control` ou `shftrort` par exemple. Lors de l'étude de l'influence d'une option comme cela a été fait pour les trois compilateurs pour les mesures quantitatives, quelques catégories présentent des variations de scores qui ne peuvent pas être associées à une seule option. La table 3.3 présente les résultats pour le compilateur `clang`. Les deux autres compilateurs ayant des résultats beaucoup plus stables, ils sont disponibles en annexe 4.5. Quelques catégories montrent des variations de score assez faibles, et ce pour toutes les options présentées ici. Les catégories présentant le plus de variations de score sont `flgctrl`, `segreg` et `bit`. Les deux dernières présentant un très faible nombre de gadgets, il est difficile de conclure sur la sensibilité du binaire apportée. Pour les gadgets de type `flgctrl` par contre, le nombre de gadgets associé est suffisamment important pour que les variations observées ne soient pas juste dues à l'échantillon. Encore une fois ici, l'écart type des variations étant très fort vis-à-vis de la moyenne implique qu'une option seule n'est pas responsable de la diminution de la sensibilité du binaire.

Lors de la conception d'un logiciel, le choix des options avec un compilateur `clang` ne peut donc pas être fait de manière incrémentale. Chaque ensemble d'options doit être testé au cas par cas pour savoir si le binaire produit contient des gadgets plus ou moins faciles à utiliser pour un attaquant. Pour le compilateur `gfree`, le choix des options influe assez peu sur la qualité des gadgets présents, à l'exception des options d'optimisation qui induisent une difficulté supplémentaire significative, notamment sur les gadgets de type `stack` et `arith`, des catégories importantes à protéger. Le compilateur `gcc` quant à lui n'influe que très peu sur la qualité des gadgets, à l'exception de la catégorie `control`.

3.1.5 Conclusion sur tar

L'étude du logiciel `tar` a permis de mettre en avant plusieurs résultats sur l'influence que peut avoir la chaine de compilation sur la sensibilité d'un logiciel face au détournement de flot d'exécution. Dans un premier temps, le choix du compilateur a une importance pour éviter une trop forte présence de gadgets utiles à un attaquant dans le binaire. Une base de code donne avec des compilateurs des résultats bien différents en quantité de gadgets. Dans un deuxième temps, en fonction du compilateur utilisé, le choix des options peut avoir des conséquences parfois importantes sur la sensibilité apparente du binaire produit. Notamment, certaines protections fonctionnent moins bien en fonction des options qui sont utilisées. Pour certains compilateurs la sécurité quantitative vis-à-vis des attaques par détournement de flot d'exécution se fait au détriment des performances.

Il a été aussi montré que certains compilateurs ont une influence importante sur la qualité des gadgets présents dans un binaire. Pour certains compilateurs, le choix des options est important pour éviter d'avoir des gadgets trop facilement utilisables pour un attaquant. Tous les compilateurs ne sont pas égaux sur le sujet. Cependant, la combinaison des métriques quantitatives et de qualité des gadgets présents permet d'avoir des indices sur les parties sur lesquelles on peut jouer pour diminuer la sensibilité d'un logiciel face au détournement de flot d'exécution.

Cette étude présente cependant plusieurs limitations. Premièrement, le corpus du logiciel est très faible. Cette étude a été réalisée sur un logiciel unique, ayant un fort historique de développement. Le logiciel a été amélioré pendant plus d'un peu plus de trente ans (la version 1.11 date de 1992), et a donc vu passer un grand nombre de développeurs, et l'introduction de nouvelles fonctionnalités ou de corrections de sécurité. Il a donc été conçu bien avant que ce type d'attaque soit connu. Pour valider les résultats obtenus sur ce logiciel, il faudrait améliorer le corpus d'étude avec d'autres logiciels écrits en utilisant le même langage. La difficulté pour faire passer le processus à grande échelle est qu'il est difficile d'interagir avec le système de *build* d'un logiciel de manière standard. Chaque logiciel présente des différences pour verrouiller des options ou en enlever. Par exemple, l'étude de l'interpréteur `python` nous a posé beaucoup de soucis dans la construction des binaires, avant d'être mise de côté. Une seconde limitation du corpus est l'ensemble des options choisies. Ici, quelques options courantes d'optimisation ont été choisies, ainsi que des options classiques utilisées dans un contexte de sécurité avec les canaris de protection de pile. D'autres options d'optimisation peuvent être considérées.

Notamment, le compilateur `gcc` propose l'option `-Os` qui n'a pas été considéré dans l'étude, mais l'option est très utilisée dans le domaine du développement logiciel à destination des systèmes embarqués. En plus de ces options, il n'est pas rare d'avoir un ensemble d'options assez large à considérer pour les applications, comme le recommande la distribution GNU/Linux RedHat à ses contributeurs pour compiler les logiciels qui utilisent `gcc`[1].

D'autres points de cette étude peuvent être améliorés. En effet, ici `tar` a été construit avec la bibliothèque C standard GNU. D'autres bibliothèques peuvent être utilisées, notamment `musl`. Cela va probablement entrainer un changement au niveau du binaire construit, et impliquer une sensibilité différente face aux attaques par détournement de flot d'exécution. De plus, dans le domaine des systèmes embarqués, il n'est pas rare d'avoir des compilateurs dédiés. Intel fournit par exemple un compilateur spécifique pour les processeurs x86_64, dérivé de `gcc`. Nous avons montré avec le cas de `clang` et `gfree` que deux compilateurs dérivés peuvent avoir des comportements quantitativement différents, l'ajout d'un compilateur autre permet une amélioration de l'analyse présentée. Ensuite, un point commun avec le chapitre précédent, et présent dans le reste de ce document, tous les compilateurs sont étudiés pour une cible GNU/Linux. Il serait intéressant d'ajouter les spécificités du compilateur C/C++ proposé par Microsoft pour son système d'exploitation. Enfin, le chapitre 4 présente les pistes pour permettre à des développeurs pour construire des logiciels moins sensibles aux attaques par détournement de flot d'exécution. Les résultats concernant les compilateurs et leurs options y sont utilisés pour mettre en avant des règles de conception.

3.2 Le langage de programmation

La section précédente montre l'influence d'options et de compilateurs sur les sources d'un unique logiciel. Lors de la conception d'un logiciel, plusieurs autres paramètres peuvent être décidés en amont du choix du compilateur, notamment le choix du langage. L'environnement de développement de logiciels sécurisés est riche en langages utilisés. Les langages ont des objectifs et paradigmes bien différents. Le C et C++ sont encore bien utilisés. ADA est encore assez populaire dans certains domaines, par exemple en avionique et systèmes critiques. D'autres langages sont poussés par certaines communautés, comme `rust` pour les langages impératifs,

1. https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc

qui commence à gagner en popularité, fortement poussé par la fondation Mozilla. Enfin, des logiciels sont développés en `golang`, notamment dans les domaines d'administration système et en réseau. Les langages fonctionnels deviennent plus utilisés, surtout pour tout ce qui touche à la sûreté de fonctionnement. Les principaux langages de cette catégorie sont `Erlang`, `Haskell` ou encore `OCaml`.

Ces langages ont des spécificités qui permettent d'en choisir un (ou plusieurs) lors du développement d'un logiciel. Plusieurs propriétés sont souvent considérées pour définir celui ou ceux qui seront utilisés. Le modèle mémoire très permissif du C et C++ font qu'ils peuvent être mis de côté au profit de langages considérés comme *memory safe*. Cette notion de protection contre les corruptions de mémoire est introduite avec certaines hypothèses d'utilisation des fonctionnalités du langage, de besoin du logiciel et du modèle de menace.

La protection définitive contre tout type de corruption mémoire n'étant pas une hypothèse réaliste, il faut aussi considérer à quel point le logiciel est sensible aux attaques par détournement de flot d'exécution. Dans cette section nous allons développer à quel point les langages entrainent une exploitabilité différente.

3.2.1 OCaml

OCaml [2] est un langage multi-paradigmes. C'est un langage principalement fonctionnel, permettant l'utilisation de paradigmes objets et impératifs. Ce langage est utilisé par exemple pour de la preuve formelle, l'assistant de preuve `coq` est écrit dans ce langage. Parmi les projets connus on trouve `infer`, un outil d'analyse statique pour différents langages, développé par `Facebook`. Le premier compilateur `rustc` était écrit en OCaml avant d'être ré-écrit en `rust`.

Le langage est dans un certain cadre *memory-safe*. L'`ANSSI`, l'Agence Nationale de la Sécurité des Systèmes d'Information, a publié un rapport sur la sécurité des langages fonctionnels [3]. Certaines fonctionnalités spécifiques permettent de contourner la protection mémoire en place, comme la possibilité de faire appel à des bibliothèques C est du partage de mémoire avec celles-ci. Dans un cadre d'utilisation classique, la mémoire est considérée comme protégée (dans le cadre du langage). Comme présenté précédemment, des attaques par canaux cachés permettent de lancer des attaques par détournement de flot d'exécution indépendamment du caractère *memory-safe* du langage. Mambretti et. al [Mam+20] montrent comment

2. https://ocaml.org
3. https://www.ssi.gouv.fr/publication/lafosec-securite-et-langages-fonctionnels/

Gadget	Nb of occurrence	Nb binaires	Occurrence moyenne
`mov rdi, qword ptr [rbx] ; call rdi`	126 549	278	455,21
`mov edi, dword ptr [rbx] ; call rdi`	125 011	278	449,68
`add edx, eax ; jmp rdx`	55 378	798	69,40
`add rdx, rax ; jmp rdx`	55 141	652	84,57
`movsxd rax, dword ptr [rdx + rax*4] ; add rdx, rax ; jmp rdx`	55 049	609	90,39
`add al, 0x82 ; add rdx, rax ; jmp rdx`	55 038	609	90,37
`add byte ptr [rax + 0x63], cl ; add al, 0x82 ; add rdx, rax ; jmp rdx`	53 843	534	100,83
`add byte ptr [rax], al ; mov rdi, qword ptr [rbx] ; call rdi`	39 665	278	142,68
`mov rdi, rbx ; call rax`	32 361	1350	23,97
`mov edi, ebx ; call rax`	32 308	1444	22,37
`add eax, edx ; jmp rax`	28 056	2689	10,43

TABLE 3.4 – Gadgets les plus présents, avec un pivot différent de `return`, En rouge les gadgets « OCaml »

utiliser l'exécution spéculative des processeurs modernes pour mettre en place du détournement de flot d'exécution.

Le langage est disponible avec un compilateur permettant de produire des binaires natifs. La communauté dispose d'un outil pour la publication et le partage des outils écrits dans le langage, permettant, entre autres, à tout utilisateur de recompiler à partir des sources un certain nombre de logiciels. Cet outil, `opam`, permet au mainteneur de contraindre la version du compilateur par exemple, pour assurer le bon déploiement d'un logiciel. Bien que certaines distributions étudiées ici contiennent des binaires écrits en OCaml, le maximum de logiciels disponibles via l'outil `opam` a été ajouté au corpus pour valider les résultats obtenus. En effet, l'architecture des binaires est atypique comparée au reste du corpus disponible, d'un point de vue détournement de flot d'exécution. Certains gadgets ne sont présents que dans des binaires écrits en OCaml. La table 3.4 illustre cette particularité. On y trouve les gadgets les plus présents dans le corpus, en étudiant les gadgets qui ne finissent pas par l'instruction `return`.

La première caractéristique des gadgets présentés qui est intéressante est l'utilisation du registre `rdi` comme registre de stockage pour l'adresse de saut. Ce registre est historiquement dédié au passage d'argument entier de fonction, bien que ce ne soit aujourd'hui plus qu'une convention d'appel. Cette convention est encore très utilisée, puisqu'il s'agit notamment de celle utilisée sur les systèmes GNU/Linux en x86_64. Ce détail n'a cependant que très peu d'influence sur la possibilité de construire une chaine de détournement de flot de contrôle. Les processeurs x86_64 se comportent de la même façon, quel que soit le registre utilisé pour un appel de fonction.

La seconde caractéristique, plus importante, est la présence abondante du

gadget dans les binaires concernés. La table 3.4 présente les gadgets les plus présents, pour tous les sauts autres que `return`. Le nombre d'occurrences des deux premiers gadgets est bien supérieur à la moyenne des autres gadgets fréquents. Affichés en rouge sur la table, ces gadgets sont cinq fois plus présents dans leur forme la plus simple que les autres gadgets fréquents. Ces gadgets récupèrent une adresse en mémoire, la mettent dans le registre `rdi` (et son équivalent 32 bits `edi`), pour ensuite faire un appel de fonction vers l'adresse contenue dans le registre.

Deux points intéressants sur ces gadgets sont observés. Le premier est que tous les binaires qui contiennent ces trois gadgets sont écrits en OCaml. Aucun autre binaire du corpus écrit dans un autre langage ne contient ce gadget. Ce gadget est aussi permissif en termes de contrôle de flot que d'autres gadgets équivalents présents dans les autres binaires. Le gadget en lui-même n'entraine donc pas une sensibilité accrue des binaires du langage. Cependant, ce gadget peut servir de signature pour donner une idée de comment un binaire a été produit. Un attaquant cherchant à savoir comment un binaire est produit pour mieux préparer son attaque va pouvoir déterminer qu'un tel binaire a été écrit en OCaml. Ensuite, tous les logiciels écrits en OCaml produisent ce gadget. Plusieurs versions du compilateur ont été utilisées, et toutes ont produit ce gadget. Il n'est donc a priori pas possible de générer un binaire avec le compilateur `ocaml.opt` sans ce gadget. Ce gadget étant très permissif sur le contrôle de flot, il peut être important à protéger. La forte occurrence du gadget dans le binaire va cependant être un frein aux méthodes de protection trop coûteuses en performance.

3.2.2 Haskell

Les résultats obtenus en étudiant les binaires issus de sources écrites en OCaml nous ont guidés vers un autre langage fonctionnel, Haskell. Cependant, cette étude est arrivée tard, et un corpus dédié à cette étude n'a pas pu être constitué. Les binaires étudiés ici sont présents sur un système qui n'a pas été conçu avec pour objectif l'analyse du langage. Seuls deux binaires ont donc été identifiés comme ayant des sources en Haskell, `gitit` et `pandoc`.

Le premier logiciel, `pandoc` est un utilitaire pour traduire un texte de certains formats vers d'autres, notamment TEX, `HTML` ou encore `markdown`. Le binaire fait 283,7 ko, contient 6144 gadgets, dont 2219 gadgets uniques. Le second logiciel, plus petit, est `gitit`. Ce logiciel est un service web permettant l'édition de wiki, permettant à la fois une saisie de wiki classique et une modification versionnée

basée sur le logiciel `git`. Le logiciel `gitit` dépend aussi de `pandoc`. Le binaire de `gitit` a une section exécutable de 25,8 ko, et comprend 829 gadgets dont `372` uniques. Les deux binaires sont construits pour le même système, Gentoo Linux x86_64.

Le corpus d'analyse est donc bien plus restreint que celui utilisé pour l'étude du langage OCaml. Le corpus est cependant suffisant pour valider le cas particulier d'OCaml. La première est de confirmer le gadget signature présenté comme étant bien une signature du langage OCaml et non du paradigme fonctionnel. Comme on a montré qu'aucun compilateur présent dans le corpus (dont celui du langage Haskell) ne générait le gadget signature d'OCaml, le détail des gadgets d'Haskell n'invalide pas ce résultat. Cependant, la présence de gadgets équivalents pour les deux binaires Haskell est intéressante. Le tableau 3.5 présente les gadgets les plus présents dans ces deux binaires. Ce tableau montre plusieurs éléments permettant de valider un comportement du paradigme fonctionnel. En effet, le gadget le plus présent dans ces deux binaires est le gadget de taille 1 suivant : `jmp qword ptr [rbx]`, gadget qui est plus présent en moyenne que le simple gadget `ret`. Le gadget est très présent dans ces deux binaires et assez proche du gadget signature d'OCaml `mov rdi, qword ptr [rbx] ; call rdi`. Cependant, contrairement à ce dernier, le gadget le plus présent dans les deux binaires aux sources en Haskell n'est pas limité à ce langage. Ce gadget est présent dans des logiciels écrits dans différents langages, et de manière fréquente. En effet, 21 binaires comptent ce gadget plus de 200 fois ce gadget, comme `Wireshark` (`C`), `Inkscape` (`C++`), `gregorio` (`OCaml`) ou encore `gnat-gps` (`ADA`). Le dénombrement utilisé ici n'est pas des plus pertinents, car, par exemple, `gitit` n'est pas dans les 21 binaires listés, car trop petit. Cela permet cependant de confirmer que le gadget est généré par beaucoup de compilateurs. Le gadget est suffisamment petit (juste une instruction pivot) pour ne pas être aussi spécifique.

Le tableau 3.5 montre aussi qu'un autre gadget est très présent, mais que dans un seul des deux binaires. En effet le gadget `add byte ptr [rax], al ; retf 0` n'est présent que dans le binaire `pandoc`. Il a la particularité d'avoir un pivot classique écrit de manière peu conventionnelle. En effet, l'instruction `retf 0` est en tout point équivalente à l'instruction `ret`. Elle précise cependant qu'aucune action pour vider la pile supplémentaire n'est réalisée. L'instruction `retf 0x10` par exemple vide la pile d'exécution de 8 octets supplémentaires à la sortie de fonction. Cela permet à un processeur de libérer de la mémoire en une seule instruction. Le gadget particulier ici a une densité non négligeable, puisqu'il est présent 0,989 fois

Gadget	Nombre d'occurrences	Nombre de binaires	Occurrence moyenne
`jmp qword ptr [rbx]`	972	2	486,00
`ret`	691	2	345,50
`add byte ptr [rax], al ; retf 0`	274	1	274,00
`retf 0`	274	1	274,00
`add byte ptr [rax], al ; add byte ptr [rax], al ; add dword ptr [rax], eax ; add byte ptr [rax], al ; retf 0`	273	1	273,00
`add byte ptr [rax], al ; add dword ptr [rax], eax ; add byte ptr [rax], al ; retf 0`	273	1	273,00
`add dword ptr [rax], eax ; add byte ptr [rax], al ; retf 0`	273	1	273,00
`retf`	82	2	41,00
`add ebp, 8 ; jmp qword ptr [rbx]`	81	2	40,50
`add rbp, 8 ; jmp qword ptr [rbx]`	81	2	40,50
`and rbx, 0xfffffffffffffff8 ; add rbp, 8 ; jmp qword ptr [rbx]`	75	2	37,50

TABLE 3.5 – Gadgets les plus présents dans les deux binaires Haskell `gitit` et `pandoc`

par ko de section exécutable. Comme pour l'autre gadget présent dans ce tableau, ce comportement n'est pas limité au compilateur du langage Haskell. Le corpus initial comporte en effet 69 binaires ayant une densité supérieure à 0,5 par ko. La majorité de ces binaires, comme `ecl`, un interpréteur et traducteur de LISP, et `perl` ont des tailles de section exécutables très faibles, en dessous de 2 ko. Seuls 13 de ces binaires ont une taille supérieure à 2 ko. Ces binaires sont `pandoc`, étudié ici, et les binaires associés aux deux logiciels `ssh-add` et `ssh-agent` sur 6 systèmes x86_64 (Arch Linux, Gentoo Linux, Ubuntu 16.04, Fedora, Debian 9, Debian Testing). Ces deux utilitaires de gestion de clef ssh sont tous les deux écrits en C, et ont une taille de section exécutable comparable aux 284 ko de `pandoc`, entre 155 ko et 181 ko.

Cette petite étude autour des deux logiciels aux sources en Haskell nous a permis de voir que tous les langages n'ont pas forcément des gadgets marquants. Cependant, l'utilisation de certains gadgets peut être supérieure à ce qui est fait en moyenne dans les autres langages et une génération de binaire peut être classée comme normale pour un compilateur et atypique pour d'autres, dans la limite de notre corpus d'analyse ici. Les résultats présentés dans cette section ainsi que la section 3.2.1 à propos du langage OCaml montrent que le choix du langage a un effet conséquent sur les différents gadgets rencontrés. Une étude plus approfondie notamment en utilisant d'autres métriques et des corpus plus importants est envisageable.

3.3 Spécificités matérielles

Cette section présente l'étude de l'influence de l'architecture matérielle sur les différentes métriques mises en place. Pour déterminer cette influence et éviter des biais possibles, les systèmes ont été construits de toute pièce pour être le plus possible similaire d'une architecture à une autre. Cela a été fait en utilisant l'outil `Buildroot` qui permet de construire des systèmes GNU/Linux complets à destination de systèmes embarqués. L'outil permet de créer une configuration unique, et en ne changeant que la cible matérielle, le système est cross-compilé depuis les sources pour la cible en question. Les systèmes sont construits pour six architectures matérielles différentes :

— i386
— x86_64
— ARM 32 bits
— ARM 64 bits
— SPARCv8 (32 bits)
— SPARC LEON (32 bit)

Bien que l'architecture LEON soit très proche de l'architecture SPARCv8, le coût de construction étant mineur, elle a été ajoutée. Bien que `Buildroot` permette de construire le système pour du SPARCv9 (64 bits), `ROPGadget` ne supportant pas l'architecture, elle a été mise de côté.

D'une manière différente, avec moins de contrôle qu'avec `Buildroot` sur les paramètres de compilation utilisés, une distribution minimale Debian 9.3 a été installée pour cinq architectures matérielles différentes :

— i386
— x86_64
— ARM 32 bits EABI (noté `el` par convention chez Debian)
— ARM 32 bits hard-float (noté `hf`)
— ARM 64 bits

Le support de l'architecture SPARC chez debian n'est pas aussi abouti que pour les autres architectures, au moment de la collecte des binaires, le processus d'installation *cross platform* n'a pas abouti.

L'architecture matérielle peut être importante pour plusieurs raisons. Tout d'abord un premier point abordé au chapitre 1 est que la possibilité de trouver des gadgets non attendus par le développeur dans le code suite à une lecture décalée. Cela étant dit, la plupart des architectures matérielles ont des compilateurs

spécifiques. Pour être plus exact, la partie traduction depuis les représentations internes va être spécifique à chaque architecture. De plus, certaines extensions permettent de prendre en compte des options particulières pour la sécurité. Certaines architectures peuvent aussi proposer des implémentations différentes de concepts similaires, comme avec les fanions `W⊕X`.

Duplication de gadgets

Pour étudier les différences entre les architectures, une métrique qui montre des résultats assez significatifs est le taux de duplication des gadgets. Ce taux a été défini au chapitre 2 comme le rapport entre le nombre total de gadgets et le nombre de gadgets uniques présents dans un binaire. Lorsque le taux vaut 1, la valeur minimale, chaque gadget n'est présent qu'une seule fois dans le binaire. Cette métrique permet notamment de savoir si la protection contre des gadgets problématiques va avoir des conséquences notables en termes de taille de code. Si un gadget n'est présent qu'une seule fois, la protection dudit gadget est, elle aussi, présente une seule fois. Comme les systèmes analysés ici sont des systèmes embarqués, la taille du binaire est un facteur assez important dans la conception.

Dans un premier temps l'analyse s'est portée sur les systèmes construits avec `Buildroot`. Les différents résultats pour le taux de duplication de gadgets sont présentés en figure 3.5. Sur cette figure, l'architecture SPARC Leon est listée, mais les points ne sont pas visibles. La raison est qu'ils sont quasiment tous confondus avec ceux de l'architecture SPARCv8. Les deux architectures étant extrêmement proches, le compilateur utilisé génère les mêmes binaires, à de très faibles différences près. Seuls quelques gadgets diffèrent de l'une à l'autre. Dans la suite de la section, l'architecture Leon n'est donc pas mentionnée explicitement et est confondue avec l'architecture SPARCv8 classique.

Le taux de duplication a déjà été utilisé pour montrer des différences entre les catégories RISC et CISC des architectures matérielles au chapitre 2 en section 2.3.1, page 54. Le détail des architectures mis de côté précédemment est mis en valeur ici. L'équilibre entre les architectures dans la section présente permet d'avoir plus de détails. On voit notamment sur la figure 3.5 que la distinction entre RISC et CISC n'est pas faisable. D'un côté, seules des architectures RISC ont un taux qui reste environnant 1 quelle que soit la taille du binaire. Mais, l'architecture ARM 64 bits se détache des autres architectures RISC pour suivre une courbe similaire aux deux architectures CISC présentes. Ces deux architectures ont cependant une

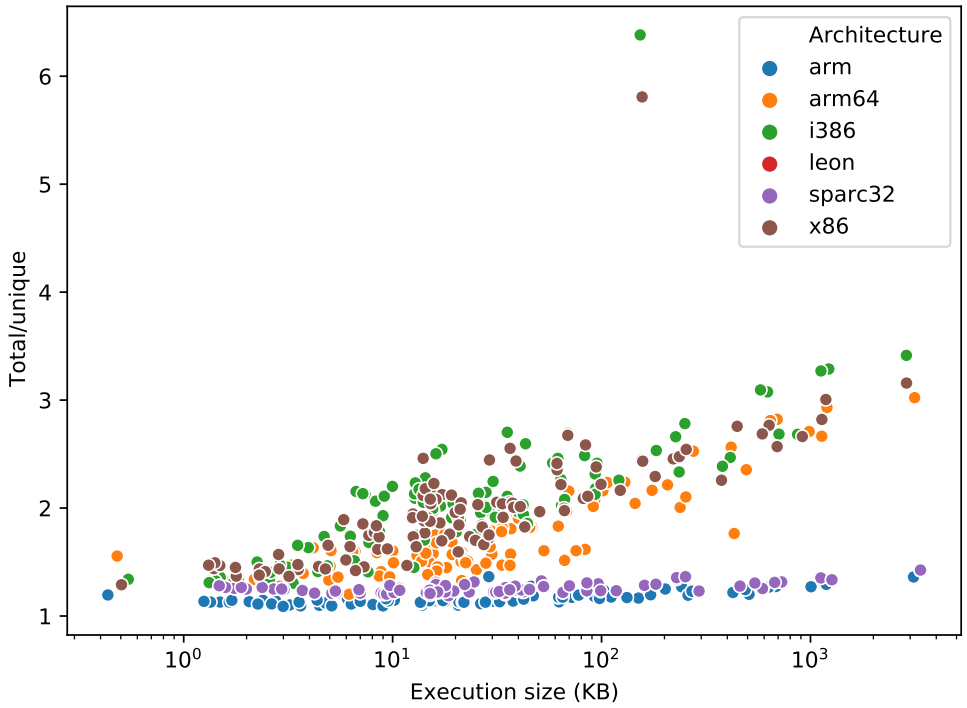


FIGURE 3.5 – Réutilisation des gadgets par binaire en fonction des architectures

valeur extrême de ratio pour le binaire `ecpg`[4] (5.8 sur x86_64 et 6.4 sur i386) que ne partage pas l'architecture ARM 64 (avec 2.04 de taux de duplication).

Les gadgets fournis sur certaines architectures matérielles sont bien plus dupliqués que sur d'autres. Un gadget est rarement présent plus d'une fois, en moyenne, dans un binaire pour une architecture RISC que dans son équivalent sur cible Intel. Lorsque l'on regarde le cas du logiciel `screen`, installé sur tous les systèmes `Buildroot`, cela est plus apparent. Bien que ce logiciel et ces résultats aient été présentés au chapitre 2 en page 55, ils sont rappelés ici en table 3.6. Le logiciel n'est pas installé sur les systèmes Debian présentés dans cette section, seuls les résultats pour les systèmes `Buildroot` sont donc présentés. Cette table présente plusieurs résultats. Le premier confirme ce que les figures précédentes montrent : le ratio de gadgets uniques sur gadgets totaux est aux alentours de 1 pour les architectures RISC, à l'exception de l'architecture ARM 64 pour laquelle le ratio est supérieur à 2 et très proche du ratio des deux architectures CISC. Au-delà

4. La description du logiciel `ecpg` est un préprocesseur embarqué SQL pour le langage C. Il convertit à la compilation les appels SQL par des appels systèmes et du C.

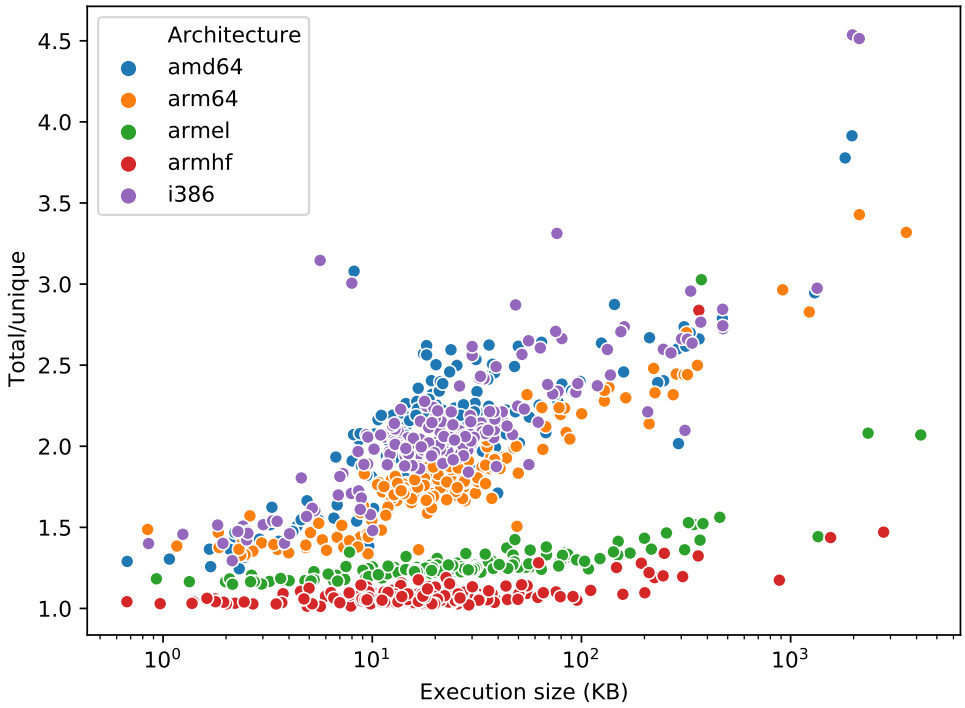


FIGURE 3.6 – Réutilisation des gadgets par binaire en fonction des architectures

de cette duplication, tous les binaires associés au logiciel ont une taille de section exécutable du même ordre de grandeur d'une architecture à l'autre. Cependant, sur cet ensemble, la taille n'est pas corrélée au nombre de gadgets trouvé dans le binaire. Par exemple, entre les deux architectures CISC i386 et x86_64, l'architecture 64 bits présente 27 % de gadgets uniques en moins, et 26 % de gadgets totaux en moins pour une taille de section exécutable égale. L'architecture ARM 64 présente elle un peu plus de gadgets uniques que son homologue 32 bits pour presque deux fois plus de gadgets totaux, pour une taille de section exécutable légèrement inférieure.

Un autre point soulevé par ce tableau est que le ratio de duplication des gadgets n'est pas un indicateur suffisant pour décrire la sensibilité d'une architecture par rapport à une autre. En effet, bien que le ratio de duplication sur les architectures CISC soit bien supérieur aux autres, la sensibilité quantitative des binaires peut varier du simple au double sans corrélation avec le ratio de duplication. Ce ratio semble cependant être un élément de distinction entre des familles d'architectures, à l'instar d'autress éléments de comparaison pour les langages de programmation.

Deux architectures matérielles ici sont disponibles à la fois en 32 bits et 64

Distribution/système	Taille de section exécutable	Nb total de gadgets	Nb de gadgets uniques	Ratio
Buildroot (x86_64)	241 ko	9843	3995	2,476
Buildroot (i386)	240 ko	13 325	5484	2,334
Buildroot (leon)	300 ko	14 190	11 529	1,231
Buildroot (SPARCv8)	300 ko	14 220	11 547	1,221
Buildroot (ARM 32)	265 ko	8508	7135	1,192
Buildroot (ARM 64)	259 ko	16 717	7950	2,103

TABLE 3.6 – Comparaison des différents mesures et éléments du programme `screen` sur les systèmes `Buildroot` du corpus d'analyse

bits, d'un côté avec i386 et x86_64, de l'autre ARM 32 et ARM 64. Les architectures n'ont pas changé que le mode d'adressage pour passer par exemple d'ARM 32 à ARM 64. Des instructions ont pu changer, être ajoutées ou supprimées. Ces changements ont des conséquences sur la génération du binaire à partir du même code source. C'est d'autant plus vrai que la partie de traduction en langage machine depuis les langages intermédiaires des compilateurs peut être faite par des équipes différentes. Ces équipes peuvent faire des choix d'instructions et d'optimisations propres à l'architecture inexistante dans l'autre. Ces changements ont donc des conséquences sur les gadgets qui sont présents dans les binaires, ce qui permet d'avoir une potentielle explication sur les observations faites au sujet du logiciel `screen`.

3.4 Conclusion

Le processus de construction d'un binaire est un mécanisme complexe à qualifier en termes de sensibilité aux attaques par détournement de flot d'exécution. Des modifications proches du processus permettent d'obtenir des métriques bien différentes. Ce chapitre a permis de mettre ces résultats. Dans un premier temps, le choix du compilateur a une influence assez forte sur la sensibilité d'un logiciel aux attaques par détournement de flot d'exécution. Il a été montré que ce choix de compilateur ainsi que des options qui les accompagnent peuvent affecter grandement les métriques mises en place dans le chapitre 2. Bien que ces résultats soient limités à un cas d'étude unique du logiciel `tar`, les résultats sont suffisamment intéressants pour valider l'utilisation des métriques.

Ces métriques ne sont cependant pas suffisantes. Cela a été montré notamment avec l'étude des binaires dont les sources sont écrites en OCaml. En effet, la présence de « gadget signature », gadget unique au langage et présent de manière

abondante dans les binaires en question, peut motiver l'ajout de cette notion à un indicateur de sensibilité qui se voudrait plus générique. Le cas de gadget signature n'a cependant été relevé pour l'instant que pour les logiciels OCaml, il serait donc intéressant d'essayer de voir si d'autres langages ou *frameworks* peuvent avoir ce même comportement.

Enfin, bien que l'étude soit très limitée, ce chapitre a brièvement présenté des distinctions notables en termes de densité et d'unicité des gadgets en fonction de la cible matérielle pour un logiciel. Le cas du logiciel `screen` a permis de mettre en valeur ces résultats.

Le chapitre suivant met en avant l'utilisation des résultats présents dans ce chapitre et dans le chapitre précédent pour proposer des éléments d'aide à la conception. Ces éléments vont permettre à des développeurs de concevoir des logiciels avec des indices sur les éléments à modifier ou choisir pour avoir une sensibilité moindre aux détournements de flot d'exécution.

Chapitre 4

Mieux concevoir un logiciel

4.1 Introduction

Les chapitres précédents ont mis en place des métriques qui permettent de déterminer en partie la sensibilité d'un logiciel face aux attaques par détournement de flot d'exécution. Ces métriques à la fois quantitatives et qualitatives ont permis de comparer différents outils, procédés et algorithmes utilisés dans la production des binaires. Ce chapitre présente des façons d'utiliser ces métriques lors de la conception pour permettre de faire des choix plus pertinents vis-à-vis du détournement de flot de contrôle. Dans un premier temps, il est question de différences que présentent deux systèmes similaires. Ensuite, il est question de l'évolution du logiciel `Firefox`, et des variations de sensibilité que cette évolution implique. Ce chapitre se termine sur une discussion autour de la reproductibilité de la construction d'un système et des conséquences que cela peut avoir sur la facilité à construire des attaques par détournement de flot d'exécution.

4.2 Différentiation de distributions

Un système Linux peut être mis à disposition d'utilisateurs de différentes manières. Les communautés gérant les distributions ont des politiques de déploiement, de choix de mise à jour des logiciels, de versions qui peuvent beaucoup varier.

Certaines distributions, comme Debian ou RedHat, ont des cycles assez longs d'une version à une autre. Des choix de version sont fixés à un moment du cycle, et

seuls les bogues sont corrigés, et l'ajout de fonctionnalités est repoussé aux versions futures. La sortie d'une version ne signifie pas l'arrêt du développement pour ces versions. Principalement, des correctifs de sécurité sont adaptés aux versions précédentes des logiciels. Pour reprendre, le logiciel `Firefox` cité plusieurs fois dans cette thèse est pour l'instant fixé en version 68 sur Debian 10. Bien que les correctifs de sécurité soient pensés pour la dernière version de `Firefox` par les équipes de Mozilla, la communauté Debian va adapter ces correctifs à la version 68. Ce principe de fonctionnement crée une version spécifique à Debian du logiciel `Firefox`. Bien qu'il soit possible à un utilisateur de reconstruire les binaires de son système, l'utilisation courante est la récupération de binaires déjà construits pour une machine semblable à celle de l'utilisateur.

À l'opposé, des distributions comme Gentoo Linux ou Arch Linux sont sur des cycles rapides de développement. Ces distributions n'ont pas de version les qualifiant, il n'existe pas d'Arch Linux 2 par exemple. Ces distributions ont un fonctionnement dit de *rolling release*. Le principe est de chercher à avoir la dernière version de chaque logiciel, au plus proche de ce que propose l'équipe de développement du logiciel. Ces distributions ont aussi l'avantage de laisser plus de contrôle à leurs utilisateurs. Gentoo Linux, par exemple, ne distribue quasiment aucun binaire, et les utilisateurs peuvent choisir les options de compilation qui seront utilisées pour leur système.

Ce concept a été décrit au chapitre 2 lors du choix des binaires analysés dans cette thèse. Cette section présente la comparaison de distributions en termes de sensibilité, surtout quantitative, en montrant les différences observées sur les logiciels distribués par les deux distributions. Dans la suite de la section, la distribution est assimilée au système construit en utilisant la distribution en question. Par exemple, le terme « la distribution Ubuntu » doit être compris comme « notre système Ubuntu ». Bien que ce soit acceptable pour des distributions de binaires, ce choix peut être plus discutable pour les distributions de sources. Les conclusions qui sont tirées de l'analyse prennent en compte cet état de fait.

4.2.1 Comparaison des distributions Fedora 26 et Ubuntu 16.04

Les deux distributions utilisées pour montrer les différences qui existent entre deux binaires d'un même logiciel sont Fedora 26 et Ubuntu 16.04. Ces deux distributions sont sélectionnées, car leurs communautés ont une philosophie proche

en termes de développement. Ce sont deux distributions de binaires, ayant un cycle de sortie assez rapide, tous les 6 mois environ. Les deux communautés n'hésitent pas à tester de nouvelles fonctionnalités, et parfois vont supprimer ces ajouts s'ils ne correspondent pas à leurs besoins. Par exemple, les équipes de Fedora ont choisi `upstart` pour remplacer `sysvinit` dans un premier temps avant de se tourner vers le plus populaire `systemd` en l'espace de deux sorties.

La distribution Fedora utilisée ici contient 1615 binaires et la distribution Ubuntu en contient 2019. Les deux systèmes ont 679 logiciels en commun, dont les binaires portent le même nom. La comparaison est faite de manière naïve, et si un logiciel appose la version au nom du binaire, alors les deux binaires ne sont pas comparés. Par exemple, `python2.7` et `python3.4` ne sont pas comparés un à un.

Les sections suivantes présentent l'utilisation des différentes métriques utilisées jusqu'à présent pour montrer les variations de sensibilités d'un même logiciel d'une distribution à une autre. Lors de cette comparaison, le système de base est le système Fedora, et l'étude porte sur les changements induits en migrant sur le système Ubuntu.

4.2.2 Variations de densité et taille de section exécutable

Le premier élément marquant lors du passage d'une distribution à l'autre porte sur les différences en termes de taille de binaire et de densité. En effet, commencer par regarder la différence en nombre absolu de gadgets n'est pas pertinent si la taille des binaires varie trop pour un logiciel. Les binaires les plus remarquables pour cette variation sont les binaires du logiciel `python2.7`. En effet, chez Fedora, l'interpréteur est limité au strict minimum et tout est déporté dans des bibliothèques partagées, là où chez Ubuntu le choix a été fait de faire un binaire monolithique. Ainsi, le binaire fait 466 octets sur le système Fedora contre 1,83 Mo sur le système Ubuntu. Bien que ce soit un cas extrême, d'autres logiciels présentent de fortes variations de taille exécutable. Sur les 679 binaires en commun, seulement 583 présentent moins de 50 % de variation de taille exécutable. Et le nombre descend peu ensuite, puisque 543 logiciels présentent des binaires avec des tailles variant de moins de 20 % d'un système à l'autre.

Prenons donc ces derniers cas, dans un premier temps, pour lesquels les métriques quantitatives sont le plus pertinentes. Le tableau 4.1 présente les différences en termes de taille, densité et nombre de gadgets des binaires en passant de Fedora à Ubuntu. Un nombre positif signifie une plus forte valeur sur le

	Taille	Taille (%)	densité (gadgets/ko)	densité (%)	Nb gadgets	Nb gadgets (%)
Moyenne	−8805 o	−1,19 %	1,50	9,3 %	61	7,6 %
Écart type	68,4 ko	6,86 %	20,5	111,0 %	3356	106,9 %
Min	−791 ko	−19,8 %	−77,1	−71,3 %	−20 894	−73,1 %
Médiane	−165 o	−1,1 %	0,1	0,5 %	−2	−1,1 %
Max	91,2 ko	19,7 %	360	1905 %	40 868	1892 %

TABLE 4.1 – Variations de propriétés des binaires de Fedora vers Ubuntu

système Ubuntu que sur le système Fedora.

Ce tableau montre trois catégories de chiffres, et chacun à la fois en absolu et en relatif. Dans les deux premières colonnes sont montrés les changements de taille de section exécutable d'un système à l'autre. Comme les binaires sélectionnés ont une variation limitée à 20 %, le résultat n'a rien de bien surprenant : peu de variations relatives de la taille de la section exécutable sont observées. Malgré cette faible variation relative, la dernière ligne présente un maximum de variation de densité (quatrième et cinquième colonnes) et du nombre de gadgets (sixième et septième colonnes) très importants. Ce maximum correspond au logiciel de coloration de graphe `edgepaint`, issu de la suite `Graphviz`. Les binaires ont des tailles quasiment identiques (117 ko et 116 ko) et présentent de très fortes différences en termes de densité et nombre de gadgets. En effet, le binaire sur Ubuntu contient presque 20 fois plus de gadgets que son homologue sur Fedora, passant de 2160 gadgets uniques à 42 028, faisant ainsi grimper la densité à 361,4 gadgets par kilo octet de section exécutable, seuls deux autres logiciels ont des augmentations à taille constante entre leurs binaires respectifs, `cluster` et `gvmap`. Les augmentations pour ces deux logiciels sont respectivement de 1090 % et 1155 %. Ces deux logiciels sont issus de la même suite que le logiciel `edgepaint` : `graphviz`. Bien que d'autres logiciels soient présents dans le paquet de distribution, ils ne présentent pas ces mêmes caractéristiques. L'explication d'une telle variation n'a pas été trouvée lors des travaux de cette thèse. L'exploration des procédés de production doit être décortiquée, et l'absence de sortie standardisée, de fichier de production comparable rend cette tâche ardue à automatiser. La mise en place de description standardisée de procédé de construction permettrait de mieux identifier d'éventuelles sources de divergence. Comme vu dans le chapitre précédent, le point important à noter est que de mêmes sources peuvent produire des binaires de sensibilité au détournement de flot d'exécution bien distinctes.

Le tableau montre que dans l'ensemble, les logiciels gardent une sensibilité quantitative plutôt stable. Pour illustrer ce propos, la figure 4.1 présente la dis-

tribution des variations de densité entre des binaires d'un même logiciel comparé respectivement. Cette figure concerne les logiciels qui ont des binaires dont la taille ne varie pas fortement (moins de 20 %) entre les deux distributions. La distribution est plutôt équilibrée, avec une moyenne de 1,29 % de changement, un écart type de 14,1 et une médiane à 0,4 %, lorsqu'on a enlevé les trois cas particuliers précédents. Les maxima de variation sont 58,2 % d'augmentation et 73,1 % de diminution. Ces chiffres sont en nombre de gadgets uniques. Ces résultats sont dans les variations attendues d'après les résultats de l'étude de compilateurs au chapitre 3. En effet, l'utilisation d'options différentes sur le logiciel `tar` a montré que le choix d'options de compilation peut grandement changer le nombre de gadgets uniques. Les variations observées entre les binaires de chacun des logiciels présents sur les deux distributions Fedora et Ubuntu permettent de voir que bien que des changements soient souvent observés, la sensibilité d'un logiciel ne varie généralement pas beaucoup avec ces choix d'options, quantitativement parlant, dans le cas où la taille de section exécutable ne varie pas de manière notable. L'utilisation de métriques quantitatives n'est pas vraiment pertinente sur le cas des binaires ayant des tailles trop différentes d'un système à l'autre. La densité est la seule métrique quantitative qui serait pertinente dans ces cas.

4.2.3 Variations de qualité de gadgets

L'étude précédente n'apporte que peu d'aide à un développeur cherchant à réduire la sensibilité de son logiciel. Il est laissé avec des options à tester pour diminuer légèrement la densité de gadgets uniques. Cette section présente l'utilisation d'une métrique qualitative qui ne va pas non plus lui permettre de faire des choix significatifs, mais qui présente des résultats intéressants pour une potentielle construction d'attaques.

Au chapitre 2, en section 2.3.2, page 61, le score d'utilisabilité d'un gadget a été défini. Le score d'un gadget commence à 0, pour un gadget n'ayant pas d'effet de bord, et augmente pour chaque instruction dans le corps du gadget ne permettant pas d'aider l'attaque. Dans le cas général, un attaquant va privilégier des gadgets à faible score pour faciliter la construction d'une chaîne de détournement de flot d'exécution. Dans cette section, un gadget pertinent est un gadget qui a un score inférieur ou égal à 0.5. Cela laisse un peu de liberté à un attaquant d'utiliser des gadgets dont le seul bruit est la modification de registres non importants (ni \%rip, ni \%rsp) et non utilisés par la première instruction du corps.

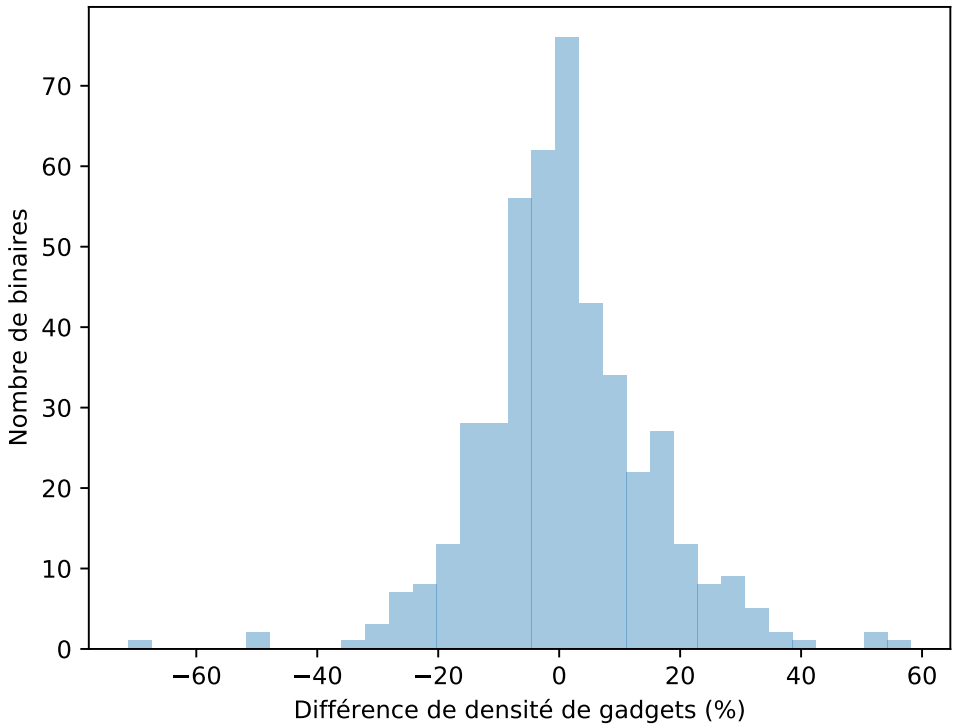


FIGURE 4.1 – Distribution de la variation de densité d'un binaire sur Fedora vers son homologue sur Ubuntu

Cette section, comme la précédente, étudie la variation observée en passant du système Fedora au système Ubuntu. Cette section présente donc l'utilisation de cette notion de gadgets pertinents sur le système Fedora. Sur les gadgets présents dans chacun des binaires, la qualité est variable. Le taux de gadgets pertinents d'un binaire est défini comme le rapport entre le nombre de ces gadgets avec un score faible au nombre de gadgets uniques dans ce binaire. L'étude porte sur les 679 logiciels en commun entre les deux systèmes. Les binaires associés à ces logiciels sur Fedora ont un taux moyen de gadgets pertinents de 66,2 %, avec très peu de variation : l'écart type vaut 5 avec des valeurs comprises entre 52,8 % et 89,9 %. Ce taux varie peu en moyenne entre les binaires des deux systèmes. La différence moyenne est de 0,7 point, avec un écart type de 4,46.

Certaines valeurs sont cependant remarquables. En effet, ce taux peut varier de −29,6 points à 25,1 points. Les deux logiciels en question sont les seuls à présenter

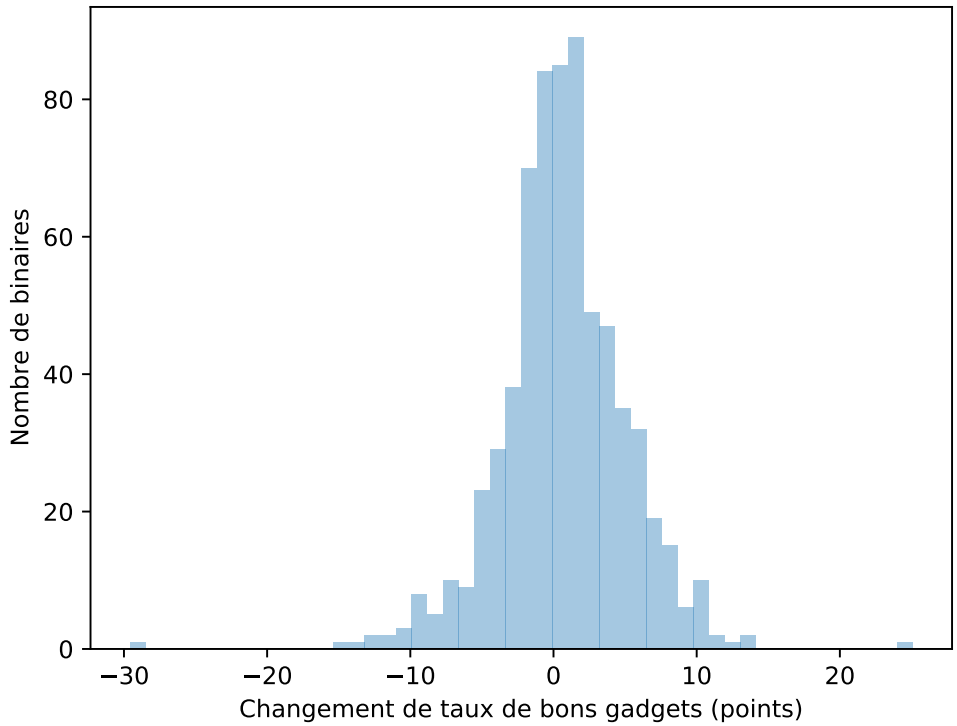


FIGURE 4.2 – Distribution de la variation de taux de bons gadgets d'un binaire sur Fedora vers son homologue sur Ubuntu

des variations de plus de 20 points. Il s'agit de `gnome-thumbnail-font` et `msgmerge` respectivement. Les deux logiciels présentent de faibles changements de taille de section exécutable, −6,2 % et 3,4 % et de fortes diminutions de densité de gadgets, −71,3 % et −49,3 % de gadgets uniques. La diminution de la densité peut donc entrainer la diminution des gadgets pertinents comme d'autres moins pertinents. Cette diminution du taux de gadgets pertinents n'est cependant pas généralisée sur tout le corpus. En effet, seulement 26 logiciels présentent une variation de plus de 10 points (dans un sens ou dans l'autre). Pour seulement deux logiciels (`ncgen3` et `mpplu`), cette variation ne s'accompagne pas d'une forte variation de la densité. En effet les 24 autres logiciels ont des variations de densité supérieures à 20 %. Le tableau en annexe 4.5 page 120 présente tous les chiffres et noms des logiciels concernés.

4.2.4 Partage de gadgets

Enfin, le dernier point de cette étude, qui est discuté dans la fin de ce chapitre, est le partage de gadgets entre des binaires d'un même logiciel, mais construits différemment. En effet, ce chapitre a montré que la densité et la quantité des gadgets changent, mais pas de manière très significative dans le cas général, et que la qualité des gadgets est stable dans l'ensemble. Cependant, les gadgets changent d'un binaire à l'autre, et cette section montre l'étendue du changement de composition des gadgets d'un logiciel. Cette section utilise le terme de gadget partagé pour les gadgets qui sont présents dans les deux binaires.

La figure 4.3 montre la distribution des gadgets partagés, toujours sur nos deux systèmes Fedora et Ubuntu. Cette figure représente un corpus de 679 binaires. Certains logiciels ont un faible partage de gadgets, dû à une grande variation de taille ou densité, comme cela est décrit dans les deux sections précédentes. Le point remarquable sur cette figure est le faible taux de partage moyen de gadgets uniques, 37,4 %, un écart type assez faible, 12,1 et une médiane très proche de la moyenne, 36,95 %. Le plus gros taux de partage de gadgets est de 74,7 % et le plus faible taux de partage, forcément biaisé par le cas de `python2.7`, est de 0,79 %. Mais même si seuls les logiciels présentant des binaires avec moins de 10 % de variation de taille de section exécutable, le minimum de partage de gadget est de 10,63 %. Si plutôt que la taille de section exécutable, la limite est le changement de densité de gadget, à 10 % aussi, le minimum de taux de partage est toujours du même ordre de grandeur, à 11,14 % de partage de gadgets uniques.

Cela permet de voir que des logiciels partageant des sources, avec quelques modifications peuvent présenter des gadgets légèrement différents. Les apports par une distribution en termes de modifications légères des sources et choix d'options de compilation vont avoir des effets légers sur la sensibilité du logiciel à la fois en termes de qualité et quantité de gadgets. Un attaquant est donc à même de connaître la difficulté à construire une chaîne pour attaquer un logiciel sur un système ou un autre. Cependant, l'attaque va généralement être limitée à un seul des deux systèmes, les gadgets utilisés pour l'un n'étant généralement pas présents sur le second système.

Cette conclusion peut cependant être améliorée. En effet, ce taux moyen de 37,4 % de partage de gadgets représente 616 gadgets. Rien ne nous permet de conclure que cela est insuffisant pour construire les attaques sur les deux systèmes. Lors d'attaques simples comme désactiver le DEP ou exécuter un `shell`, peu de

gadgets sont nécessaires pour que l'attaque réussisse. Il faudrait donc regarder plus en détail, au cas par cas la liste des gadgets partagés, binaire par binaire pour vérifier si les gadgets utiles sont tous dedans. Le processus n'est cependant pas très automatisable et demande une forte présence humaine. L'outil utilisé pour lister les gadgets dans les binaires, ROPgadget [Sal12] propose cependant de construire automatiquement une chaîne de gadgets permettant l'ouverture d'un `shell`. Pour ce faire, l'outil cherche une liste précise de gadgets pour construire la chaîne. L'absence d'un seul gadget dans cet ensemble restreint de gadgets fait échouer l'outil. Sur l'ensemble des binaires i386 et x86_64, soit 11 303 binaires, l'outil ne réussit à générer des chaînes que pour 56 binaires, soit 0,5 %. L'outil ne permet pas de générer ces chaînes pour d'autres architectures matérielles. Il n'est cependant pas impensable de pouvoir fournir un outil permettant de vérifier si plusieurs systèmes partagent suffisamment de gadgets pour construire une telle chaîne.

Autres comparaisons

Bien que cette section se soit concentrée sur l'étude de la comparaison des deux systèmes Fedora 26 et Ubuntu 16.04, d'autres comparaisons ont été faites. Toutes les distributions GNU/Linux présentes dans le corpus ont été comparées au moins une fois, même si toute la combinatoire n'a pas été réalisée. 9 comparaisons ont été faites, et la comparaison Fedora-Ubuntu est l'une d'entre elles. Des résultats similaires ont été observés, validant ainsi la méthode et les résultats observés dans ce cas développé dans cette section.

4.3 Reproductibilité d'un *build*

La tendance actuelle de certaines distributions GNU/Linux, notamment Debian, est la mise en avant de la reproductibilité d'un *build*. Debian distribue des binaires via des dépôts, et a une politique orientée sur la disponibilité des sources à l'origine de ces binaires. D'un point de vue utilisateur, jusqu'à récemment, il fallait faire confiance à Debian pour la correspondance entre les sources disponibles et le binaire mis à disposition. Cette relation de confiance ne suffit pas aux contributeurs de Debian, qui veulent qu'on puisse vérifier que le binaire récupéré corresponde aux sources disponibles. C'est ce qui a conduit à la mise en place des *builds* reproductibles. À quelques détails près identifiés, un utilisateur peut reconstruire

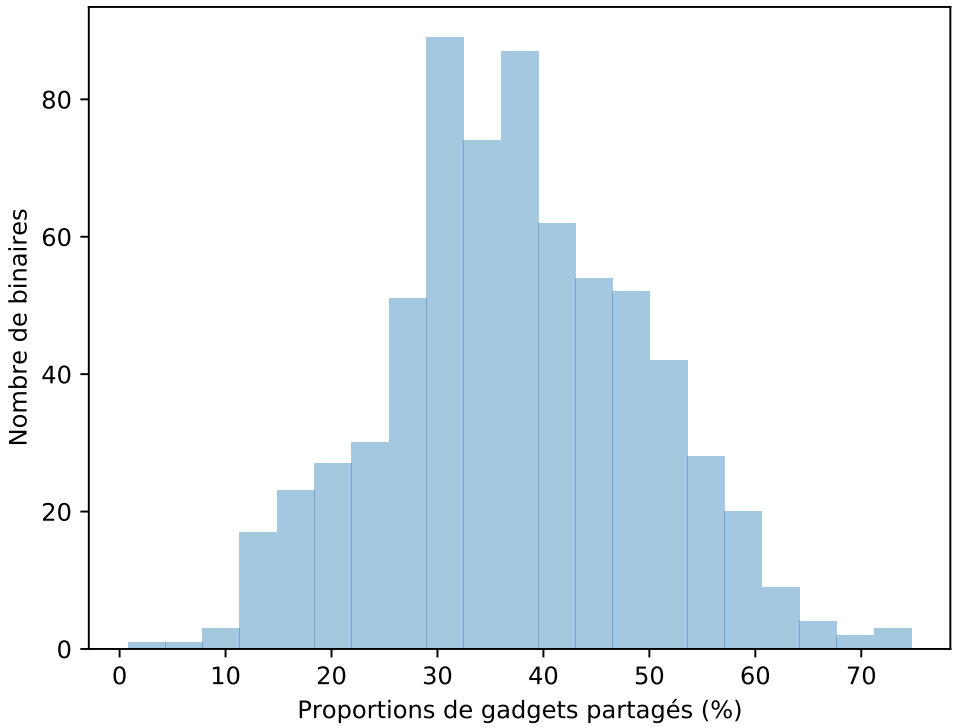


FIGURE 4.3 – Distribution du partage de gadgets uniques entre un binaire de Fedora et son homologue sur Ubuntu

le binaire et vérifier qu'il correspond à celui mis à disposition sur les dépôts. Les détails identifiés qui varient dans ces *builds* reproductibles sont par exemple les chemins sur le système de fichier vers le dossier de *build*.

Cette notion de *build* reproductible a des conséquences intéressantes dans le cadre de construction d'attaque par détournement de flot d'exécution. Dans le modèle de menace présenté au premier chapitre, cela n'est pas le cas, car l'attaquant a une connaissance absolue du binaire qu'il attaque. Cependant, ce modèle peut être restreint. Un attaquant peut par exemple connaître les caractéristiques du système qu'il attaque, mais n'a pas accès exactement à la configuration. Ce modèle a du sens dans le cas d'un équipementier qui fournit des systèmes sur mesures bien documentés. Deux clients peuvent avoir des systèmes proches, et l'un deux peut étudier son système, récupérer des informations comme les logiciels présents et les numéros de version de ces logiciels.

La variation de compilation d'un binaire à un autre peut permettre de produire des binaires suffisamment différents pour empêcher un attaquant ayant un binaire (parmi un groupe de binaires) associé au logiciel de construire efficacement une attaque. Cela introduit un coût supplémentaire de conception d'attaque si l'attaque concerne un grand nombre de binaires du même logiciel. L'approche habituelle des systèmes GNU/Linux de distribution de binaires laisse donc dans ce modèle de menace plus d'informations aux attaquants sur les éléments disponibles pour construire en amont les chaines de détournement de flot d'exécution, comme le permet ROPgadget par exemple. La reproductibilité stricte n'est donc peut-être pas souhaitable dans certaines situations. La reproductibilité du processus, donnant un binaire différent mais garantissant un ensemble de propriétés de sécurité pourrait être un but plus souhaitable.

4.4 Intégration continue et suivi de sensibilité

L'objectif principal de ces métriques est de pouvoir fournir à un développeur des outils qui lui permettent d'évaluer la sensibilité d'un logiciel aux attaques par détournement de flot d'exécution. Certains choix peuvent être faits en amont du projet, comme le langage. D'autres vont être imposés, et certaines métriques permettent de choisir un compilateur de manière éclairée pour remplir des objectifs de sécurité.

Les métriques mises en place dans les chapitres précédents ont permis de mettre en évidence qu'une approche purement *security by design* n'est pas possible. La génération du binaire d'un logiciel est un processus complexe dont la sensibilité résultante ne peut pas être déterminée a priori. Cette thèse a cependant montré qu'il est possible d'automatiser l'évaluation de la sensibilité d'un binaire aux attaques par détournement de flot d'exécution. Il est donc possible d'intégrer l'évaluation de cette sensibilité à un processus d'intégration continue. De la même manière que le fonctionnement du logiciel est testé à chaque nouvelle évolution, des métriques peuvent être utilisées pour permettre aux développeurs de déterminer si les derniers changements sur les sources ont une influence positive ou négative sur la sensibilité du logiciel.

Les travaux effectués dans cette thèse ne montrent que la faisabilité d'un tel système, cependant aucun outil n'est fourni, un travail d'ingénierie supplémentaire doit être fourni pour permettre un déploiement aisé d'une telle procédure. Pour

montrer la faisabilité d'un tel système, ce chapitre finit sur l'étude de l'évolution de la sensibilité du logiciel `Firefox` aux attaques par détournement de flot d'exécution.

4.4.1 Cas d'étude : `Firefox`

La section 4.2 a mis en avant les différences métriques d'un logiciel, en moyenne, d'une distribution binaire vers une autre. Cette comparaison présente cependant deux paramètres à évaluer et éviter. Le premier est que le compilateur et ses options peuvent varier de l'une à l'autre. Les résultats montrés au chapitre 3 on mis en avant l'influence forte que ces choix provoquent sur la sensibilité d'un logiciel aux attaques par détournement de flot d'exécution. Le second paramètre est la composition des équipes, voire du mainteneur unique, d'une distribution à une autre. Les mainteneurs peuvent avoir des vues différentes sur la manière de construire un logiciel. Pour notre cas d'étude, le logiciel `Firefox`, le choix a donc été de se concentrer sur les binaires distribués par la fondation Mozilla directement. Le logiciel est donc étudié dans son environnement de développement initial.

L'étude porte donc sur l'évolution du logiciel à chacune des versions distribuées, de la première version disponible en 64bits à la dernière version sortie au moment de l'étude, la version 65.0b9. Cela ne représente pas la granularité la plus fine possible du développement du logiciel, mais cela permet une visualisation assez intuitive des évolutions. L'ensemble représente 829 instances du logiciel. Cette granularité du corpus ne permet pas d'expliquer des changements de sensibilité. Cela permet cependant de justifier la faisabilité d'un système automatique pour évaluer l'influence de changements.

La métrique utilisée pour ce cas d'étude est la densité de gadgets uniques par ko de section exécutable. Bien que cette métrique ne soit pas la meilleure [BP19], elle présente déjà des propriétés intéressantes, comme cela a été montré plusieurs fois dans cette thèse. La figure 4.4 montre les résultats de densité au cours du temps. Cette figure présente plusieurs points montrant le potentiel intérêt du suivi de la sensibilité lors du développement d'un logiciel. Le premier est que des groupes bien distincts en termes de densité sont présents. Dans les versions récentes, à droite de la figure, trois groupes de binaires sont assez distants les uns des autres, avec des points regroupés au sein de chaque ensemble. Bien que les écarts soient présents sur des *releases* majeures (version 52.7, version 54 et version 57), sans explications précises, la variation entre deux sauts est loin d'être négligeable. Ensuite, l'histoire du logiciel avant ces versions montre une densité ayant des variations autour de 65

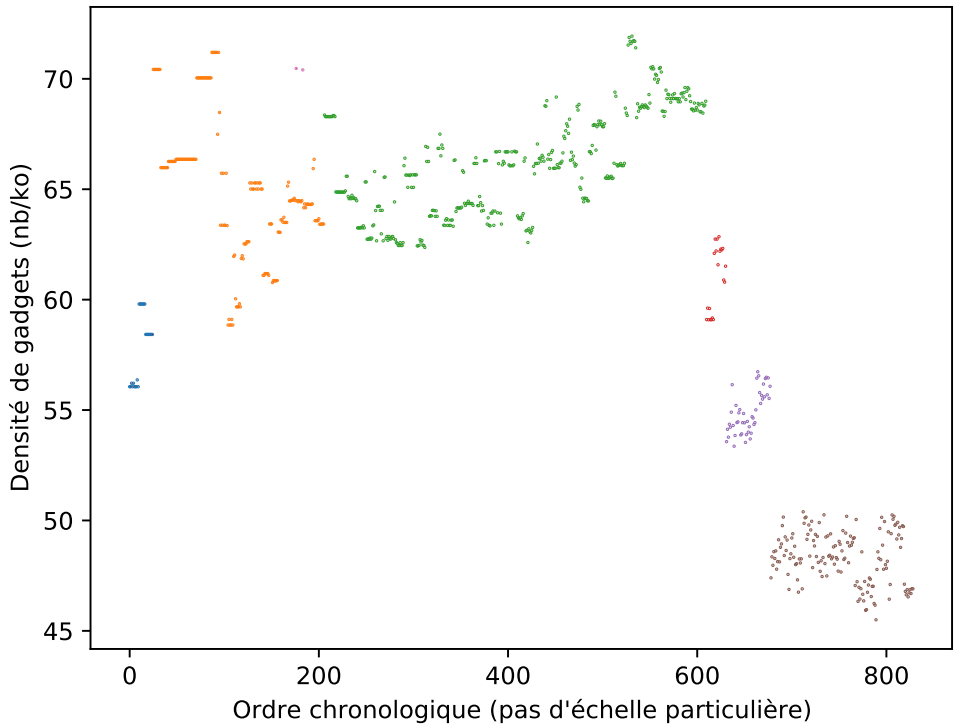


FIGURE 4.4 – Densité de gadgets uniques par ko de section exécutable, dans l'ordre des versions du logiciel `Firefox`

gadgets par ko de section exécutable, en légère augmentation. Bien qu'on puisse déterminer des *clusters* avec des algorithmes type dbscan dans ces versions, rien ne semble lier des versions majeures ou mineures à des augmentations de la densité.

L'évaluation de la sensibilité d'un seul binaire ne représente pas une grosse charge. En effet, créer un ensemble de comparaison pour un logiciel unique, ici `Firefox` ne prend pas beaucoup de temps. Il faut compter environ une minute par binaire pour générer les données nécessaires pour toutes les métriques quantitatives. L'outil ne passe pas très bien à l'échelle pour le moment, et sur des ensembles plus grands, il a tendance à ralentir. Pour de l'analyse au jour le jour, dans un environnement d'intégration continue, où seul un binaire est ajouté à l'ensemble, cela n'est pas limitant.

4.5 Conclusion

Les métriques définies dans cette thèse peuvent servir à évaluer la sensibilité d'un logiciel au cours de son développement. Ce chapitre a permis de mettre en avant l'utilisation de métriques pour qualifier l'évolution de la sensibilité d'un logiciel, que ce soit au cours de son développement ou lors des modifications pour sa distribution.

L'utilisation des métriques au cours du développement d'un logiciel peut permettre une meilleure compréhension des raisons qui font qu'un logiciel est plus ou moins sensible aux attaques par détournement de flot d'exécution. Les métriques présentées dans cette thèse et utilisées dans ce chapitre sont prometteuses pour améliorer le processus de production de binaire à des fins de sécurité en détournement de flot d'exécution.

Conclusion

Lors de la conception d'un logiciel, la sécurité peut être prise en compte suffisamment en amont pour ne plus être vue comme une contrainte tardive de conception, mais une fonctionnalité à part entière. Lorsque la sécurité d'un logiciel est prise en compte suffisamment tôt, cela va influer sur les décisions de technologies, architecture et sur l'environnement de conception du logiciel. Cette approche est souvent nommée *security by design*. Cette thèse cherche à apporter des outils permettant d'appliquer une telle démarche dans le contexte d'attaques par détournement de flot d'exécution.

Afin de mettre à disposition de tels outils, dans un premier temps cette thèse présente l'environnement dans lequel se situe un logiciel. Ce qu'est un logiciel a été défini avec les éléments qui le composent. Bien que l'objet d'étude de cette thèse soit le logiciel et sa conception, pour définir et comprendre les attaques par détournement de flot d'exécution, un logiciel est associé dans cette thèse aux fichiers binaires qui le composent. L'objet principal responsable de cette transformation est le compilateur. Cet outil est simplifié dans cette thèse. Il réalise la transformation de sources d'un logiciel en binaire en deux étapes, la traduction en langage de description interne, et la génération du code. Le flot d'exécution d'un logiciel est défini par un développeur au niveau des sources, et ces transformations modifient ce flot. Pour étudier et comprendre les attaques par détournement de flot d'exécution, la modélisation du flot retenue dans cette thèse est le flot d'exécution du binaire, après les différentes étapes de transformation.

Cette thèse décrit le fonctionnement des attaques par détournement de flot de contrôle et les éléments permettant la construction de ces attaques : la corruption mémoire permettant l'entrée et les gadgets constituant l'attaque. La suite du chapitre 1 présente un ensemble de protections existantes permettant de limiter l'efficacité des attaques. Les protections présentées ne sont pas exhaustives, mais

permettent d'avoir une idée des différents principes importants utilisés pour se prémunir de ces attaques : la limitation du nombre de gadgets, la dissimulation d'informations permettant de trouver ces gadgets et la vérification que le flot d'exécution correspond à un modèle calculé lors de la compilation. La sensibilité résultante après l'application des protections est difficilement comparable. Il n'existe dans la littérature que peu de moyens de comparer différentes protections. De plus, afin de permettre l'identification des causes de sensibilité, des métriques génériques de sensibilité face aux attaques par détournement de flot d'exécution sont nécessaires. L'absence de telles métriques dans la littérature ou la nécessité d'avoir une forte contribution manuelle pour les obtenir a motivé les travaux présentés dans ce document.

Dans le chapitre 2, cette thèse propose plusieurs métriques pour mesurer la sensibilité de logiciels à des attaques par détournement de flot d'exécution. Dans un premier temps des métriques de sensibilités quantitatives sont présentées, en commençant par un simple dénombrement des gadgets présents dans un binaire. Cette métrique est étendue en la normalisant par la taille de la section exécutable. Ces deux métriques sont insuffisantes pour qualifier la sensibilité globale d'un binaire. Ce chapitre présente ensuite rapidement une métrique de diversité peu exploitée, et enchaîne sur la présentation d'une métrique qualitative. Cette métrique est basée sur le score de facilité d'utilisation d'un gadget, introduite par Follner et al. [FBB16].

Pour mettre en place ces métriques de sensibilité, j'ai choisi de me baser sur l'étude de binaires présents sur des systèmes existants. Cette approche a permis de collecter un grand nombre de binaires, de différentes provenances, permettant une analyse statistique approfondie. La caractérisation du système d'étude est présentée aussi au chapitre 2. Enfin, ce chapitre a permis de valider la caractérisation syntaxique des gadgets, en introduisant une forme minimale d'écriture de gadgets. Grâce à cette forme, nous avons montré que dans aucun de nos binaires, deux gadgets syntaxiquement différents ne produisent pas exactement le même résultat.

Cette thèse présente dans le chapitre 3 la complexité du processus de construction d'un binaire d'un point de vue sensibilité aux attaques par détournement de flot d'exécution. Cette complexité est illustrée au travers d'un cas d'étude, le logiciel `tar`. Le procédé de construction du logiciel a été modifié pour être maitrisé dans son intégralité. Cette étude a permis de mettre en avant l'influence du compilateur et des options utilisées sur la sensibilité d'un binaire d'un même logiciel. Cette étude montre aussi l'efficacité d'une protection contre le détournement de flot d'exécution

sur l'architecture matérielle x86, protection déployée dans un compilateur modifié. Pour continuer dans une approche de *security by design*, d'autres paramètres décidés en amont de la conception sont étudiés dans ce chapitre, le langage et l'architecture matérielle cible. Les spécificités de deux langages fonctionnels, OCaml et Haskell, sont ainsi montrées.

Enfin, le chapitre 4 illustre l'utilisation possible des métriques pour qualifier la sensibilité potentielle d'un système aux attaques par détournement de flot d'exécution. Il en ressort notamment que des modifications mineures d'un logiciel ou du processus de création du binaire associé permettent de rendre l'attaque plus difficile dans un environnement où l'attaquant est moins puissant. L'idée développée est que la reproductibilité de la construction de binaires fait fuiter des informations qui peuvent être dissimulées avec un processus plus aléatoire. La philosophie de *build* reproductible est mise en avant par des distributions GNU/Linux comme Debian. Ce chapitre montre aussi qu'une approche plus continue de la sensibilité d'un logiciel est possible, au travers de l'étude de cas de l'évolution du logiciel `Firefox`.

Perspectives

Plusieurs éléments d'amélioration des travaux sont possibles. Le premier est que malgré la validation de plusieurs métriques de sensibilité, aucun indice général n'a pu être défini pour qualifier la sensibilité d'un logiciel aux attaques par détournement de flot d'exécution. L'objectif de comparer la sensibilité de deux logiciels ou même de deux binaires doit se faire sur plusieurs axes hétérogènes, ce qui n'est pas toujours très facile à manipuler.

Ensuite, bien qu'une métrique qualitative soit définie pour les architectures i386 et x86_64, les autres architectures matérielles peuvent avoir une telle métrique. Les autres architectures présentes dans cette thèse ont un jeu d'instructions limité, car sont des architectures RISC. Étendre cette métrique est une tâche envisageable, à court terme. La généralisation de cette métrique permettra d'avoir une comparaison plus facile de la sensibilité des binaires entre les différentes architectures.

Cette limite sur les métriques est due aux choix initiaux d'outils pour l'étude de gadgets, qui se sont montrés insuffisants pour ce genre d'étude. ROPgadget permet de récupérer la liste des gadgets dans un assembleur associé à l'architecture désignée

dans le binaire. Un autre outil BAP [1] permet de désassembler des binaires dans un langage intermédiaire indépendant de l'architecture du binaire. Le formalisme présenté par cet outil permet aussi de faire une analyse statique des binaires plus approfondie que ce qui a été fait dans cette thèse. Cela permettrait aussi d'étendre l'étude d'équivalence des gadgets, et éventuellement proposer un outil similaire à un compilateur permettant de construire des chaines à partir d'une charge à exécuter et d'un binaire cible.

Bien que l'objectif principal de cette thèse est l'aide à la qualification d'un processus de conception pour produire des logiciels moins sensibles au détournement de flot d'exécution, aucun logiciel simple d'utilisation n'a été produit. Un trop grand nombre d'opérations manuelles sont nécessaires pour évaluer la sensibilité de nouveaux logiciels au système d'étude. Un travail d'ingénierie supplémentaire est nécessaire pour rendre la suite d'outils utilisable de façon engageante. L'intégration des métriques à un outil d'intégration continue comme *sonarqube* est une approche à envisager pour améliorer le retour vers l'utilisateur. De plus, les métriques ont mis en évidence des résultats intéressant lors de la comparaison d'un ou plusieurs binaires à un ensemble de binaires. Il serait aussi souhaitable qu'un outil d'intégration continue qui prend en compte la sensibilité des attaques par détournement de flot d'exécution puisse donner des résultats de comparaisons vis-à-vis d'une base de données générique.

Dans un objectif à plus long terme, les travaux dans cette thèse peuvent être étendus à des thématiques de protections. Une meilleure compréhension d'où vient la sensibilité d'un logiciel aux détournements de flot d'exécution permet de construire des outils de protection plus performants. Cela peut aussi permettre le développement d'un outil de conseil ou de configuration automatique d'une chaine de développement. Avec certains paramètres de constructions fixés, par exemple les sources et un compilateur, un outil pourrait conseiller les paramètres optimaux pour la protection ou pointer les fonctions les plus critiques à protéger ou sur lesquelles travailler pour limiter la sensibilité. Cette thèse propose des éléments pour arriver à ce résultat, mais la granularité des métriques utilisées est trop globale, et ne permet pas de pointer des fonctions, ou les zones des sources responsables de la sensibilité, en produisant par exemple une carte de chaleur de sensibilité.

Cette thèse a notamment mis en évidence un besoin de traçabilité de construction de binaire. Afin de permettre à un développeur de mieux identifier les points d'attention, un tel outil faciliterait la tâche. Dans cette optique, il est envisageable,

1. https://github.com/BinaryAnalysisPlatform/bap

comme ce qui est fait par *godbolt*[2] et *RMSbolt*[3], de mettre en place une trace forte entre les sources et le binaire. Ces outils ont un autre but, et ne conviennent pas pour les gadgets décalés par exemple. Une traçabilité forte permet de choisir entre modifier les sources ou modifier le procédé de construction du binaire pour diminuer la sensibilité.

2. https://godbolt.org/
3. https://gitlab.com/jgkamat/rmsbolt

Annexes

Comparaisons des compilateurs gcc et gfree

Option	average	bit	inout	datamov	arith	branch	shftrot	conver	break	logical	flgctrl	segreg	stack	control
-O1	0.1(0.1)	0.1(0.4)	0.1(0.1)	0.2(0.0)	0.2(0.3)	0.1(0.3)	0.1(0.0)	0.1(0.2)	0.3(0.2)	0.1(0.0)	-0.0(0.1)	0.0(0.8)	-0.0(0.0)	-0.5(0.4)
-O2	0.1(0.1)	0.0(0.0)	0.1(0.1)	0.3(0.1)	0.2(0.2)	0.1(0.3)	0.1(0.1)	0.1(0.2)	0.2(0.3)	0.1(0.0)	0.1(0.1)	0.3(1.0)	-0.0(0.0)	-0.8(0.4)
-O3	0.0(0.1)	0.2(0.6)	-0.0(0.1)	0.3(0.1)	0.1(0.1)	-0.0(0.2)	0.1(0.1)	0.1(0.2)	0.1(0.2)	0.1(0.0)	-0.1(0.1)	0.2(0.9)	-0.1(0.0)	-0.9(0.3)
-flto	-0.0(0.0)	0.1(0.5)	0.0(0.1)	-0.0(0.0)	-0.1(0.1)	-0.1(0.1)	-0.0(0.1)	-0.0(0.2)	0.1(0.3)	-0.0(0.0)	0.0(0.1)	-0.4(1.0)	-0.0(0.0)	-0.2(0.3)
-fstack-check	-0.0(0.0)	-0.0(0.5)	-0.0(0.1)	-0.0(0.0)	0.0(0.0)	-0.0(0.0)	-0.0(0.0)	0.0(0.2)	-0.0(0.3)	-0.0(0.0)	0.0(0.1)	0.2(0.7)	-0.0(0.0)	-0.0(0.1)
-fstack-protector	0.0(0.0)	-0.1(0.7)	-0.0(0.1)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	-0.1(0.2)	-0.0(0.2)	-0.0(0.0)	0.0(0.0)	0.1(0.8)	-0.0(0.0)	-0.0(0.1)
-fstack-protector-all	0.1(0.1)	-0.2(0.6)	-0.2(0.1)	-0.1(0.1)	0.3(0.2)	0.4(0.3)	-0.1(0.1)	-0.0(0.2)	-0.2(0.3)	-0.1(0.0)	-0.0(0.1)	-0.1(0.9)	-0.0(0.0)	-0.2(0.2)
-fstack-protector-strong	0.0(0.0)	-0.2(0.6)	-0.1(0.1)	-0.1(0.0)	0.0(0.0)	0.1(0.0)	-0.0(0.0)	-0.1(0.1)	-0.1(0.3)	-0.0(0.0)	0.0(0.0)	-0.1(1.0)	0.0(0.0)	-0.1(0.1)

TABLE 4.2 – Influence de chacune des options utilisées pour le compilateur `gcc`, en terme de score pour chacune des catégorie, sur le logiciel `tar` en version 1.30

Option	average	bit	inout	datamov	arith	branch	shftrot	conver	break	logical	flgctrl	segreg	stack	control
-O0	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)
-O1	0.6(0.0)	0.0(0.0)	-0.1(0.2)	-0.0(0.0)	0.9(0.0)	0.2(0.0)	0.0(0.0)	-0.0(0.2)	-0.1(0.1)	-0.1(0.0)	-0.5(0.1)	-1.0(1.9)	0.5(0.0)	0.0(0.2)
-O2	0.6(0.0)	0.0(0.0)	0.2(0.1)	-0.0(0.0)	0.9(0.0)	0.2(0.0)	0.0(0.0)	-0.3(0.5)	0.0(0.2)	-0.1(0.1)	-0.6(0.1)	-1.0(1.9)	0.5(0.0)	0.1(0.2)
-O3	0.6(0.0)	0.0(0.0)	0.1(0.1)	-0.0(0.0)	0.9(0.0)	0.2(0.0)	0.1(0.0)	0.1(0.4)	-0.1(0.2)	-0.1(0.0)	-0.4(0.1)	-1.0(1.9)	0.5(0.0)	-0.0(0.1)
-fno-stack-protector	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)
-fstack-protector	-0.0(0.0)	0.0(0.0)	-0.0(0.2)	-0.0(0.0)	0.0(0.0)	-0.0(0.0)	0.0(0.0)	0.3(0.4)	-0.3(0.1)	0.0(0.0)	0.0(0.1)	0.0(0.0)	-0.0(0.0)	-0.0(0.2)
-fstack-protector-all	-0.0(0.0)	0.0(0.0)	0.0(0.1)	-0.0(0.0)	0.0(0.0)	-0.0(0.0)	0.0(0.0)	0.4(0.2)	-0.2(0.2)	-0.0(0.0)	-0.0(0.1)	1.5(2.1)	0.0(0.0)	0.0(0.2)
-fstack-protector-strong	-0.0(0.0)	0.0(0.0)	-0.1(0.2)	-0.0(0.1)	-0.0(0.0)	-0.0(0.0)	0.0(0.0)	0.1(0.2)	-0.2(0.2)	0.0(0.1)	-0.1(0.1)	0.0(0.0)	-0.0(0.0)	0.0(0.2)
-fstrict-vtable-pointers	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)

TABLE 4.3 – Influence de chacune des options utilisées pour le compilateur `gfree`, en terme de score pour chacune des catégorie, sur le logiciel `tar` en version 1.30

Taux de bons gadgets entre Fedora et Ubuntu

Nom	gadgets partagés	%	Δgadgets	%	Δdensité	%	ΔBESS	%	Δratio bon gadgets	ratio fedora
gnome-thumbnail-font	72	13.26	-397	-73.1	-77.1	-71.3	-317	-6.2	-29.6	89.9
ncgen3	298	28.63	-64	-6.1	-1.1	-4.4	-789	-1.8	-11.0	78.5
hcitool	355	14.72	547	22.7	8.7	22.6	43	0.1	10.3	73.2
ocamlopt.opt	6077	10.37	-29213	-49.9	-6.0	-29.3	-848357	-29.0	-13.1	70.9
teckit_compile	32	55.17	1645	2836.2	-12.0	-41.1	99419	4887.9	10.3	62.1
l2ping	259	13.00	607	30.5	15.3	30.4	27	0.1	10.3	76.2
gsettings-data-convert	44	41.12	-10	-9.3	-4.0	-20.4	771	13.9	11.1	67.3
ccxxmake	86	43.88	16455	8395.4	4.5	39.8	1057075	5975.6	-13.6	82.1
mpplu	105	55.56	9178	4856.1	-1.1	-6.7	605635	5209.3	-10.3	81.0
profcheck	69	26.34	5375	2051.5	4.3	30.5	296051	1548.9	-12.6	80.5
msgmerge	74	13.86	-254	-47.6	-14.6	-49.3	620	3.4	25.1	56.4
dispwin	224	42.11	15781	2966.4	2.7	20.7	1000779	2441.3	-11.7	79.7
cluster	807	21.85	40266	1090.3	205.7	1186.7	-16341	-7.5	12.8	68.9
uuidgen	37	46.84	23	29.1	10.2	20.5	116	7.1	11.7	57.0
xsetwacom	148	26.71	-178	-32.1	-9.2	-31.3	-221	-1.1	10.4	61.9
setleds	34	40.00	-9	-10.6	2.7	10.5	-637	-19.1	13.8	61.2
cs2cs	40	18.18	-111	-50.5	-15.5	-50.4	-5	-0.1	-14.8	88.2
printcal	161	45.22	8773	2464.3	4.3	37.9	570123	1759.5	-11.8	81.2
edgepaint	572	26.48	40868	1892.0	360.0	1905.0	-757	-0.6	13.3	68.5
gvmap	797	22.78	40414	1155.3	202.6	1256.1	-16501	-7.4	10.4	71.2

Bibliographie

[Aba+09] Martin Abadi, Mihai Budiu, Úlfar Erlingsson et Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In : *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), p. 1-40. DOI : 10.1145/1609956.1609960. URL : https://doi.org/10.1145/1609956.1609960.

[AAS20] Sameed Ali, Prashant Anantharaman et Sean W Smith. "Armor Within : Defending against Vulnerabilities in Third-Party Libraries". In : *Sixth Language-theoretic Security (LangSec) IEEE Security & Privacy Workshop*. 2020.

[ANS] ANSSI. *Sécurité et langage Java*. URL : https://www.ssi.gouv.fr/agence/publication/securite-et-langage-java/.

[Ban+16] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Jason Reeves, Sean W Smith, Anna Shubina, Maxwell Koo et Michael E Locasto. "Sections are Types, Linking is Policy : Using the Loader Format for Expressing Programmer Intent". In : *BlackHat USA* (2016).

[Bat74] Michael Batty. "Spatial entropy". In : *Geographical analysis* 6.1 (1974), p. 1-31.

[Bit+14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières et Dan Boneh. "Hacking Blind". In : *2014 IEEE Symposium on Security and Privacy*. Mai 2014, p. 227-242. DOI : 10.1109/SP.2014.22.

[BL05] Sandrine Blazy et Xavier Leroy. "Formal verification of a memory model for C-like imperative languages". In : *International Conference on Formal Engineering Methods*. Springer. 2005, p. 280-299.

[BP19] Michael D Brown et Santosh Pande. “Is less really more ? towards better metrics for measuring security improvements realized through software debloating”. In : *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. Santa Clara, CA : USENIX Association, août 2019. URL : https://www.usenix.org/conference/cset19/presentation/brown.

[Buc+08] Erik Buchanan, Ryan Roemer, Hovav Shacham et Stefan Savage. “When Good Instructions Go Bad : Generalizing Return-oriented Programming to RISC”. In : *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS ’08. Alexandria, Virginia, USA : ACM, 2008, p. 27-38. DOI : 10.1145/1455770.1455776.

[Bur+17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler et Mathias Payer. “Control-Flow Integrity : Precision, Security, and Performance”. In : *ACM Comput. Surv.* 50.1 (avr. 2017). ISSN : 0360-0300. DOI : 10.1145/3054924. URL : https://doi.org/10.1145/3054924.

[CW14] Nicholas Carlini et David Wagner. “ROP is Still Dangerous : Breaking Modern Defenses”. In : *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, p. 385-399.

[CBK12] Varun Chandola, Arindam Banerjee et Vipin Kumar. “Anomaly detection for discrete sequences : A survey”. In : *IEEE Transactions on Knowledge and Data Engineering* 24.5 (2012), p. 823-839.

[Cha+14] Oscar Chaparro, Gabriele Bavota, Andrian Marcus et Massimiliano Di Penta. “On the Impact of Refactoring Operations on Code Quality Metrics”. In : *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, p. 456-460.

[CBG17] X. Chen, H. Bos et C. Giuffrida. “CodeArmor : Virtualizing the Code Space to Counter Disclosure Attacks”. In : *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. Avr. 2017, p. 514-529. DOI : 10.1109/EuroSP.2017.17.

[Che+18] Yurong Chen, Shaowen Sun, Tian Lan et Guru Venkataramani. “Toss : Tailoring online server systems through binary feature customization”. In : *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. 2018, p. 1-7.

[CFC15] Thomas Coudray, Arnaud Fontaine et Pierre Chifflier. *Picon : Control Flow Integrity on LLVM IR*. 2015. URL : https://www.sstic.org/2015/presentation/control_flow_integrity_on_llvm_ir/.

[DKW10] Thomas Dullien, Tim Kornau et Ralf-Philipp Weinmann. "A Framework for Automated Architecture-independent Gadget Search". In : *Proceedings of the 4th USENIX Conference on Offensive Technologies*. Washington, DC : USENIX Association, 2010.

[FBB16] Andreas Follner, Alexandre Bartel et Eric Bodden. "Analyzing the Gadgets". In : *Engineering Secure Software and Systems*. Sous la dir. de Juan Caballero, Eric Bodden et Elias Athanasopoulos. Cham : Springer International Publishing, 2016, p. 155-172. ISBN : 978-3-319-30806-7.

[FS01] Michael Frantzen et Michael Shuey. "StackGhost : Hardware Facilitated Stack Protection". In : *USENIX Security Symposium*. T. 112. 2001.

[Fra+18] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen et Ahmad-Reza Sadeghi. "IMIX : In-Process Memory Isolation EXtension". In : *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD : USENIX Association, 2018, p. 83-97.

[Gök+14] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos et Georgios Portokalidis. "Size does matter : Why using gadget-chain length to prevent code-reuse attacks is hard". In : *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association. 2014, p. 417-432.

[GMM16] Daniel Gruss, Clémentine Maurice et Stefan Mangard. "Rowhammer. js : A remote software-induced fault attack in javascript". In : *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, p. 300-321. DOI : 10.1007/978-3-319-40667-1_15.

[He+20] Wenjian He, Sanjeev Das, Wei Zhang et Yang Liu. "BBB-CFI : Lightweight CFI Approach Against Code-Reuse Attacks Using Basic Block Information". In : *ACM Trans. Embed. Comput. Syst.* 19.1 (fév. 2020). ISSN : 1539-9087. DOI : 10.1145/3371151. URL : https://doi.org/10.1145/3371151.

[Hen00] J. L. Henning. “SPEC CPU2000 : measuring CPU performance in the New Millennium”. In : *Computer* 33.7 (2000), p. 28-35.

[Hen06] John L Henning. “SPEC CPU2006 benchmark descriptions”. In : *ACM SIGARCH Computer Architecture News* 34.4 (2006), p. 1-17.

[Heo+18] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo et Mayur Naik. “Effective program debloating via reinforcement learning”. In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, p. 380-394.

[His+12] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall et J. W. Davidson. “ILR : Where’d My Gadgets Go ?” In : *2012 IEEE Symposium on Security and Privacy*. Mai 2012, p. 571-585. doi : 10.1109/SP.2012.39.

[HTI97] Mei-Chen Hsueh, Timothy K Tsai et Ravishankar K Iyer. “Fault injection techniques and tools”. In : *Computer* 30.4 (1997), p. 75-82. issn : 1558-0814. doi : 10.1109/2.585157.

[Jac+14] Emily R Jacobson, Andrew R Bernat, William R Williams et Barton P Miller. “Detecting code reuse attacks with a model of conformant program execution”. In : *International Symposium on Engineering Secure Software and Systems*. Springer. Springer, Cham, 2014, p. 1-18. isbn : 978-3-319-04896-3. doi : 10.1007/978-3-319-04897-0_1.

[JPL20] Hyerean Jang, Moon Chan Park et Dong Hoon Lee. “IBV-CFI : Efficient fine-grained control-flow integrity preserving CFG precision”. In : *Computers & Security* 94 (2020). issn : 0167-4048. doi : 10.1016/j.cose.2020.101828.

[Jel+20] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min et Yeongjin Jang. “MARDU : Efficient and Scalable Code Re-Randomization”. In : *Proceedings of the 13th ACM International Systems and Storage Conference*. SYSTOR ’20. Haifa, Israel : Association for Computing Machinery, 2020, p. 49-60. isbn : 9781450375887. doi : 10.1145/3383669.3398280. url : https://doi.org/10.1145/3383669.3398280.

[Jia+08] Yue JIANG, Bojan CUKI, Tim MENZIES et Nick BARTLOW. "Comparing Design and Code Metrics for Software Quality Prediction". In : *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*. PROMISE '08. Leipzig, Germany : Association for Computing Machinery, 2008, p. 11-18. DOI : 10.1145/1370788.1370793.

[Jos06] Lou JOST. "Entropy and diversity". In : *Oikos* 113.2 (2006), p. 363-375.

[Jun+20] Dongjae JUNG, Minsu KIM, Jang JINSOO et Kang Brent BYUNGHOON. "Value-Based Constraint Control Flow Integrity". In : *IEEE Access* 8 (2020), p. 50531-50542.

[KPP12] AD KEROMYTIS, Michalis POLYCHRONAKIS et V PAPPAS. "Smashing the gadgets : Hindering return-oriented programming using in-place code randomization". In : *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, p. 601-615.

[Kim+14] Yoongu KIM et al. "Flipping Bits in Memory without Accessing Them : An Experimental Study of DRAM Disturbance Errors". In : *SIGARCH Comput. Archit. News* 42.3 (juin 2014), p. 361-372. ISSN : 0163-5964. DOI : 10.1145/2678373.2665726. URL : https://doi.org/10.1145/2678373.2665726.

[Koc+18] Paul KOCHER et al. "Spectre Attacks : Exploiting Speculative Execution". In : *arXiv preprint arXiv :1801.01203* (2018).

[Kur+13] Anil KURMUS et al. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring." In : *NDSS*. 2013.

[Kuz+18] Volodymyr KUZNETZOV, László SZEKERES, Mathias PAYER, George CANDEA, R. SEKAR et Dawn SONG. "Code-Pointer Integrity". In : *The Continuing Arms Race : Code-Reuse Attacks and Defenses*. Association for Computing Machinery et Morgan & Claypool, 2018, p. 81-116. ISBN : 9781970001839. URL : https://doi.org/10.1145/3129743.3129748.

[Lu+11] Kangjie LU, Dabi ZOU, Weiping WEN et Debin GAO. "DeRop : Removing Return-Oriented Programming from Malware". In : *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC '11. Orlando, Florida, USA : Association for Computing Machinery, 2011, p. 363-372. ISBN : 9781450306720. DOI : 10.1145/2076732.2076784. URL : https://doi.org/10.1145/2076732.2076784.

[Mam+20] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, Wil Robertson, Engin Kirda et Anil Kurmus. "Bypassing memory safety mechanisms through speculative control flow hijacks". In : *arXiv preprint arXiv :2003.05503* (2020).

[MO81] Bernard Marchand et Alain Ozan. "Méthodes mathématiques de classification en géographie". In : *L'Espace géographique* (1981), p. 1-14.

[Mau+20] N. Maunero, P. Prinetto, G. Roascio et A. Varriale. "A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems". In : *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. Avr. 2020, p. 1-10. doi : 10.1109/DTIS48698.2020.9081314.

[Mil19] Matt Miller. "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape". In : Santa Clara, CA : USENIX Association, août 2019.

[Mur+14] Mark Murphy, Per Larsen, Stefan Brunthaler et Michael Franz. "Software Profiling Options and Their Effects on Security Based Diversification". In : *Proceedings of the First ACM Workshop on Moving Target Defense*. MTD '14. Scottsdale, Arizona, USA : Association for Computing Machinery, 2014, p. 87-96. isbn : 9781450331500. doi : 10.1145/2663474.2663485. url : https://doi.org/10.1145/2663474.2663485.

[Ona+10] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti et Engin Kirda. "G-Free : defeating return-oriented programming through gadget-less binaries". In : *Proceedings of the 26th Annual Computer Security Applications Conference*. 2010, p. 49-58.

[PLB18] Jevgenja Pantiuchina, Michele Lanza et Gabriele Bavota. "Improving Code : The (Mis) Perception of Quality Metrics". In : *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, p. 80-91.

[PTV09] Vit Paszto, Pavel Tucek et Vit Vozenilek. "On spatial entropy in geographical data". In : *Proceedings of the GIS Ostrava, Ostrava, Czech Republic* (2009), p. 25-28.

[PF16] Olgierd Pieczul et Simon N. Foley. “Runtime Detection of Zero-Day Vulnerability Exploits in Contemporary Software Systems”. In : *Data and Applications Security and Privacy XXX : 30th Annual IFIP WG 11.3 Conference, DBSec 2016, Trento, Italy, July 18-20, 2016. Proceedings*. Sous la dir. de Silvio Ranise et Vipin Swarup. Cham : Springer International Publishing, 2016, p. 347-363. doi : 10.1007/978-3-319-41483-6_24.

[Por+20] Chris Porter, Girish Mururu, Prithayan Barua et Santosh Pande. “BlankIt Library Debloating : Getting What You Want Instead of Cutting What You Don’t”. In : *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK : Association for Computing Machinery, 2020, p. 164-180. isbn : 9781450376136. doi : 10.1145/3385412.3386017.

[PR12] M. Prandini et M. Ramilli. “Return-Oriented Programming”. In : *IEEE Security Privacy* 10.6 (nov. 2012), p. 84-87. issn : 1558-4046. doi : 10.1109/MSP.2012.152.

[Qin+19] Peng Qin, Cheng Tan, Lei Zhao et Yueqiang Cheng. “Defending against ROP Attacks with Nearly Zero Overhead”. In : *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, p. 1-6.

[QPY18] Anh Quach, Aravind Prakash et Lok Yan. “Debloating software through piece-wise compilation and loading”. In : *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, p. 869-886.

[Roe+12] Ryan Roemer, Erik Buchanan, Hovav Shacham et Stefan Savage. “Return-Oriented Programming : Systems, Languages, and Applications”. In : *ACM Transactions on Information and System Security* 15.1 (mar. 2012), 2 :1-2 :34. issn : 1094-9224. doi : 10.1145/2133375.2133377.

[RH97] Linda H Rosenberg et Lawrence E Hyatt. “Software quality metrics for object-oriented environments”. In : *Crosstalk journal* 10.4 (1997), p. 1-6.

[RHS98] Linda Rosenberg, Ted Hammer et Jack Shaw. “Software metrics and reliability”. In : *9th international symposium on software reliability engineering*. 1998.

[Sal12] Jonathan Salwan. *ROPgadget tool*. http://shell-storm.org/project/ROPgadget. 2012. (Visité le 10/10/2019).

[Sch+15] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi et Thorsten Holz. "Counterfeit object-oriented programming : On the difficulty of preventing code reuse attacks in C++ applications". In : *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, p. 745-762. DOI : 10.1109/SP.2015.51.

[SAB11] Edward J Schwartz, Thanassis Avgerinos et David Brumley. "Q : Exploit Hardening Made Easy". In : *USENIX Security Symposium*. 2011, p. 25-41.

[Sch+20] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari et Stephanie M Schwartz. "A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks". In : *Proceedings of the ACM Conference on Computer and Communications Security*. 2020.

[Sha07] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone : Return-into-libc Without Function Calls (on the x86)". In : *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA : ACM, 2007, p. 552-561. ISBN : 978-1-59593-703-2. DOI : 10.1145/1315245.1315313.

[Sha+18] Hashim Sharif, Muhammad Abubakar, Ashish Gehani et Fareed Zaffar. "TRIMMER : application specialization for code debloating". In : *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, p. 329-339.

[Sot07] Alexander Sotirov. "Heap feng shui in javascript". In : *Black Hat Europe* 2007 (2007), p. 11-20.

[Sta+02] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou et Georgios L. Bleris. "Code quality analysis in open source software development". In : *Information Systems Journal* 12.1 (2002), p. 43-60.

[Sze+13] L. Szekeres, M. Payer, T. Wei et D. Song. "SoK : Eternal War in Memory". In : *2013 IEEE Symposium on Security and Privacy*. Mai 2013, p. 48-62. DOI : 10.1109/SP.2013.13.

[VM17] Petr Vytovtov et Evgeny Markov. "Source code quality classification based on software metrics". In : *2017 20th Conference of Open Innovations Association (FRUCT)*. IEEE. 2017, p. 505-511.

[Wan+20] Pei WANG, Jinquan ZHANG, Shuai WANG et Dinghao WU. "Quantitative Assessment on the Limitations of Code Randomization for Legacy Binaries". In : *5th IEEE European Symposium on Security and Privacy*. 2020.

[WB16] Xueyang WANG et Jerry BACKER. "SIGDROP : Signature-based ROP Detection using Hardware Performance Counters". In : *arXiv preprint arXiv :1609.02667* (2016).

[Wei] Florian WEIMER. *Recommended compiler and linker flags for GCC*. https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc. (Visité le 10/10/2019).

[Xia+16] Yuan XIAO, Xiaokuan ZHANG, Yinqian ZHANG et Radu TEODORESCU. "One Bit Flips, One Cloud Flops : Cross-VM Row Hammer Attacks and Privilege Escalation". In : *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX : USENIX Association, août 2016, p. 19-35. ISBN : 978-1-931971-32-4. URL : https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao.

[Zha+13] Chao ZHANG, Tao WEI, Zhaofeng CHEN, Lei DUAN, Laszlo SZEKERES, Stephen MCCAMANT, Dawn SONG et Wei ZOU. "Practical Control Flow Integrity and Randomization for Binary Executables". In : *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA : IEEE Computer Society, 2013, p. 559-573. DOI : 10.1109/SP.2013.44.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Sensibilité de logiciels au détournement de flot de contrôle

Mot clés : détournement de flot de contrôle, sécurité à la conception, sensibilité aux attaques, analyse de binaires

Résumé :

La sécurité d'un logiciel peut être prise en compte dès la conception. Cette approche, appelée *security by design*, permet d'influer au plus tôt de la conception pour influer sur l'architecture d'un logiciel. Les protections contre les attaques par détournement de flot d'exécution, comme le *return oriented programming* , ne sont pas pensées pour changer la manière de concevoir un logiciel, mais permettent de protéger un logiciel soit lors de sa compilation soit en travaillant directement sur le binaire produit.

Dans cette thèse, nous proposons des métriques permettant à un développeur d'évaluer la sensibilité d'un logiciel face aux attaques par détournement de flot d'exécution. Pour aider le développement, les métriques définies permettent d'identifier les paramètres de production de binaires d'un logiciel qui entrainent une sensibilité accrue à ces attaques. L'utilisation de ces métriques sont illustrées dans cette thèse en étudiant l'influence de compilateurs et de leurs options, de langages et architectures matérielles.

Title: Software sensitivity to control flow hijack

Keywords: control flow hijack, security-by-design, sensitivity to attacks, binary analysis

Abstract: The security of a software can be taken into account right from the design stage. This approach, called *security by design*, allows to influence as early as possible the design to influence the architecture of a software. The protections against control flow hijacks, such as return oriented programming, are not designed to change the way of designing the software. They often aim to protect a software either during its compilation or by working directly on the binary produced.

In this thesis, we propose metrics allowing a developer to evaluate the sensitivity of a software program to attacks by control flow hijacks. To ease development, metrics defined allow to identify the parameters used in the production of binaries of a software that result in increased sensitivity to these attacks. The use of of these metrics are illustrated in this thesis by studying the influence of compilers and their options, languages and hardware architectures.