HAL open science

Analyse et optimisation dynamiques de programmes au format binaire pour la cybersécurité

Camille Le Bon

To cite this version:

Camille Le Bon. Analyse et optimisation dynamiques de programmes au format binaire pour la cybersécurité. Cryptographie et sécurité [cs.CR]. Université Rennes 1, 2022. Français. NNT : 2022REN1S044 . tel-03906421v2

HAL Id: tel-03906421

https://theses.hal.science/tel-03906421v2

Submitted on 19 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

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

Par
Camille LE BON

Analyse et optimisation dynamiques de programmes au format binaire pour la cybersécurité

Thèse présentée et soutenue à Rennes, le 5/07/2022
Unité de recherche : Inria, Centre Inria Rennes-Bretagne Atlantique (Inria-Rennes)

Rapporteurs avant soutenance :

Karine HEYDEMANN	Maître de Conférence à Sorbonne Université
Philippe CLAUSS	Professeur à l'Université de Strasbourg

Composition du Jury :

Examinateurs :	Aurélien FRANCILLON	Professeur à EURECOM
	Guillaume HIET	Maître de Conférence à CentraleSupelec
	Frédéric TRONEL	Maître de Conférence à CentraleSupelec
	Vincent NICOMETTE	Professeur à INSA Toulouse
Dir. de thèse :	Erven ROHOU	Directeur de recherche, Inria Rennes-Bretagne Atlantique

Table of Contents

Introduction **5**

1 État de l'art **9**

1.1 Les attaques par corruption de mémoire 9

1.1.1 Les attaques par corruption de code 11

1.1.2 Les attaques sur les données 12

1.1.3 Les fuites d'informations 15

1.1.4 Les attaques par détournement du flot de contrôle 16

1.2 L'intégrité du flot de contrôle 19

1.2.1 *Control-Flow Integrity* (CFI) basés sur les sources 22

1.2.2 CFI basés sur le code binaire 23

1.2.3 L'isolation des données de contrôle 25

1.3 La modification dynamique de binaire 27

1.3.1 La *Dynamic Binary Modification* (DBM) dans la sécurité 29

1.4 Limites de l'état de l'art 30

1.4.1 Notre contribution 31

2 Damas : Un framework d'injection de protection à l'exécution **33**

2.1 Sorry : notre bibliothèque de DBM 35

2.1.1 Philosophie et fonctionnalités 36

2.1.2 Lectures et écritures via les *tampons* 38

2.2 Control-Data Isolation à l'exécution 40

2.2.1 Contraintes spatiales dans le binaire originel 41

2.2.2 Désassemblage et rebasement en mémoire 42

2.2.3 Traductions des instructions 43

2.3 Suppression des branchements indirects 46

2.3.1 Les tobbogans de Control-Data Isolation 48

2.3.2 Les classes d'équivalence de branchements 51

2.3.3 Les tables de délégation 54

2.3.4 Les appels et retours de fonctions contournant la *Procedure Linkage Table* (PLT) 67
2.4 Optimisation des tables de délégation 72
2.4.1 Un tri par utilisation décroissante 73
2.4.2 Une représentation en arbre binaire 74

3 Évaluation **77**
3.1 Évaluation de sécurité 77
3.2 Évaluation des performances 82
3.2.1 Protocole expérimental 82
3.2.2 Résultats 85

4 Travaux futurs **93**
4.1 Tirer un meilleur profit de Rust 93
4.2 Meilleure couverture des programmes existants 96
4.3 Optimisations du code actuel 98
4.4 Réduction de la taille des classes d'équivalence 101

Conclusion **103**

Bibliography **107**

Glossaire **107**

Introduction

Les programmes que nous utilisons au quotidien sont majoritairement écrits par des humains. Malheureusement, *l'erreur est humaine* et il est commun pour le programmeur d'introduire des bugs dans les programmes qu'il écrit. Ces bugs prennent des formes aussi diverses qu'étonnantes, allant du comportement imprévisible et déroutant d'une fonctionnalité toute entière du programme, à la modification involontaire d'un emplacement mémoire passant inaperçue. Si les bugs les plus visibles sont facilement détectés et donc corrigés, les plus sournois sont plus difficiles à déclencher et leur effet n'est pas forcément visible du point de vue de l'utilisateur. Malheureusement, le caractère spectaculaire d'un bug n'est en rien un indicateur de sa gravité. Bien que certains bugs ne soient pas immédiatement visibles par l'utilisateur, ils peuvent avoir des conséquences désastreuses sur la sûreté ou la cohérence du système. C'est-à-dire que le programme peut modifier des données du système hôte ou encore effectuer des opérations qu'il n'était pas censé effectuer, comme afficher des informations sensibles ou les modifier. Par exemple, si un bug introduit une importante fuite de mémoire, l'utilisateur ne le verra probablement pas tout de suite, néanmoins, au fur et à mesure de l'exécution du programme, la mémoire que ce dernier gaspille aura un impact croissant sur les performances du système tout entier, jusqu'à possiblement l'empêcher complètement de fonctionner.

Par ailleurs, il est possible d'exploiter ces bugs afin de détourner le programme et d'en tirer profit. Un programme qui permet des lectures ou écritures en mémoire à des emplacements que le programmeur n'a pas prévu est vulnérable face aux *attaques par corruption de mémoire*. Ces attaques sont particulièrement dangereuses et peuvent permettre de prendre complètement le contrôle du processus cible, le forçant à exécuter du code arbitraire décidé par l'attaquant (Szekeres et al. 2013). Ces attaques prennent de nombreuses formes en fonction de la manière dont le programme est détourné. Il est par exemple possible de provoquer des fuites de données sensible en modifiant l'adresse d'un pointeur que le programme est censé afficher, tout comme il est possible de forcer le programme à exécuter une portion de code en modifiant des données de contrôle, le poussant à prendre un branchement plutôt qu'un autre. Le classement 2021 des Common Weakness Enumeration *(CWE)* de l'organisation américaine MITRE place ces attaques

parmi les plus dangereuses, avec les écritures en dehors des bornes en première position et les lectures en dehors des bornes en troisième position (MITRE 2021).

Naturellement, de nombreuses recherches ont été effectuées afin de classifier et documenter précisément ce type d'attaques et pouvoir s'en prémunir efficacement (Abadi et al. 2005 ; Shoshitaishvili et al. 2016 ; Szekeres et al. 2013). Toutefois, malgré la diversité et la richesse des approches utilisées pour contrer ces attaques, elles restent un problème encore aujourd'hui. En effet, les approches de défense contre les attaques par corruption de mémoire souffrent toutes de limitations : leur impact sur les performances, leur manque de précision, des faiblesses inhérentes au modèle d'attaque envisagé, le besoin de recompiler le programme ou encore, l'impossibilité de greffer une protection sur un processus en cours d'exécution. De plus, les attaquants découvrent en permanence de nouvelles manières de contourner ces protections, forçant les défenseurs à se réinventer sans cesse. Cette course à l'armement a pour conséquence de devoir faire un compromis entre performances et sécurité. Or ce compromis devrait être renégocié sans cesse par l'utilisateur en fonction des risques encourus et de ses ressources.

Par exemple, si une entreprise propose un service informatique à ses nombreux utilisateurs, elle peut vouloir ajuster le niveau de protection mis en place dans ses programmes pour s'assurer de la qualité de service de ces derniers. Typiquement, si l'entreprise acquiert du nouveau matériel plus puissant, elle peut vouloir renforcer la sécurité sur ces programmes sans impact négatif sur les performances de ceux-ci par rapport à avant. À l'inverse, si son infrastructure est réduite et que la qualité de service s'en trouve impactée, elle peut vouloir privilégier les performances de ses services, au détriment de la sécurité, pour en assurer le bon fonctionnement et ne pas pénaliser ses utilisateurs. Un autre exemple serait une entreprise sous le feu d'attaques sur ses services. Cette entreprise pourrait préférer améliorer la sécurité de ses programmes au détriment de la qualité de service, puisque les protections mise en place habituellement pourraient ne plus suffire face à l'intérêt soudain d'utilisateurs malveillants pour leur système.

Malheureusement, selon les protections considérées, il n'est pas possible de renégocier ce compromis entre performances et sécurité. Par exemple lorsque cette protection nécessite une recompilation du programme et que le code source n'est pas fourni. De plus, même si une protection se base sur le code binaire du programme, elle nécessite que le processus soit relancé pour être mise en place. Cette contrainte réduit drastiquement la capacité de ces protections à s'adapter aux besoins de l'utilisateur sur le moment. En effet certains programmes peuvent être longs à démarrer ou à être arrêtés. Le fait de devoir relancer

un processus peut donc être un frein à l'adoption d'une mesure de sécurité. Par ailleurs, il est possible que les besoins de l'utilisateur changent dans le temps. Si par exemple une précédente attaque a été détectée par l'utilisateur, il peut vouloir augmenter la protection de ses services au détriment des performances de ceux-ci. À l'inverse, si l'utilisateur estime qu'il est possible de sacrifier un peu de sécurité pour plus de performances, il serait souhaitable de pouvoir renégocier ce compromis à tout moment sans relancer tout le programme. Par conséquent, nous nous sommes intéressés à cette problématique et avons cherché à développer une approche capable de s'adapter aux besoins de l'utilisateur pendant l'exécution du processus cible, que l'on souhaite protéger.

Nous avons mis au point une approche permettant de protéger une processus *à chaud* (c'est-à-dire, en cours d'exécution) contre les attaques sur son flot de contrôle. Notre approche se base sur le code binaire du programme et ne nécessite ni recompilation du programme, ni redémarrage du processus en cours d'exécution. Notre approche utilise la DBM pour instrumenter le processus cible et les informations disponibles à l'exécution pour compléter celles tirées du fichier binaire du programme, permettant la mise en place d'une protection du flot de contrôle du processus. Nos travaux ont fait l'objet d'une publication dans une conférence internationale (Le Bon et al. 2021) présentant les contributions suivantes : un framework de protection de processus à chaud ainsi qu'un outil de protection contre les attaques sur le flot de contrôle utilisant ce framework. Dans ce présent manuscript nous présenterons aussi la bibliothèque de DBM que nous avons implémentée afin de développer notre framework.

L'organisation de ce manuscrit est la suivante. Le chapitre 1 donne une définition plus précise des attaques par corruption de mémoire et présente une revue de l'état de l'art des approches d'attaques par corruption de mémoire et des défenses face à celles-ci. Nous portons une attention particulière aux attaques par détournement du flot de contrôle dont notre approche cherche à se prémunir. Dans le chapitre 2, nous décrivons précisément notre approche, les difficultés rencontrées et les solutions trouvées pour parvenir à l'outil que nous avons developpé durant cette thèse de doctorat. Dans le chapitre 3, nous exposons les résultats de nos expérimentations. Nous évaluons la sécurité apportée par notre approche ainsi que l'impact de celle-ci sur les performances du processus protégé. Enfin, nous discutons dans le chapitre 4 des possibilités d'amélioration de notre approche tant d'un point de vue conceptuel que d'un point de vue technique. Nous explicitons les limites de notre approche ainsi que de son implémentation actuelle et nous donnons des pistes pour repousser ces limites.

Chapitre 1

État de l'art

Les attaques par corruption de mémoire sont des attaques complexes et variées mais puissantes . Elles peuvent donner à l'attaquant la possibilité de modifier complètement le comportement d'un processus ciblé dans l'objectif de lui faire exécuter des portions de code arbitraires. Dans ce chapitre, nous décrivons en détail le fonctionnement des attaques par corruption de mémoire en section 1.1 afin d'en saisir toute la complexité et comprendre pourquoi il existe un nombre si grand de contre-mesures. Dans la section 1.2, nous nous intéressons plus en détails à l'intégrité du flot de contrôle. Puis nous présentons la modification dynamique de binaire en section 1.3 et son utilisation dans la mise en place de mesures de sécurité. Enfin, nous parlons des limites de l'état de l'art actuel face aux attaques par corruption de mémoire en section 1.4.

1.1 Les attaques par corruption de mémoire

Les attaques par corruption de mémoire sont des attaques consistant à tirer profit d'un bug du programme cible sur ses accès à la mémoire. D'après Szekeres et al. 2013, il suffit, pour déclencher une attaque, que l'attaquant prenne le contrôle d'un pointeur afin d'effectuer une lecture ou une écriture arbitraire dans l'espace d'adressage.

Une des techniques les plus simples et répandues pour corrompre un pointeur est le dépassement de tampon (*buffer overflow*). Il s'agit de profiter de l'absence de tests de bornes lors d'écriture dans un tampon afin d'écrire plus loin dans la mémoire que dans le tampon prévu à cet effet. C'est typiquement ce qui se produit lors de l'appel à des fonctions dépréciées de la bibliothèque C qui ne vérifient pas les bornes des tampons dans lesquels elles écrivent. Le listing 1 est une fonction vulnérable à un dépassement de tampon :

```c
void prompt_and_print() {
  char buffer[20];
  gets(buffer);
  printf("%s\n", buffer);
}
```

Listing 1: Fonction affichant le tampon donné en entrée

Dans l'exemple précédent, la fonction `gets`[1] ne teste pas les bornes du tampon dans lequel elle écrit. Par conséquent, dans cet exemple, si l'utilisateur écrit plus de 20 caractères, la fonction continuera d'écrire dans la pile et écrasera les données qui y sont stockées. Par exemple, dans le listing 2, la variable `granted` est nulle. Si l'exécution du programme se déroule normalement, elle reste nulle et la fonction `privileged_function` n'est jamais appelée. Néanmoins, si l'utilisateur écrit plus de 10 caractères et fait en sorte que l'un des caractères entre le 11e et le 14e soit non nul[2], la variable `granted` ne sera plus nulle et donc le flot de contrôle du programme sera dévié. On suppose toutefois pour cet exemple que le compilateur respecte l'ordre de déclaration des variables pour faciliter l'explication, ce qui n'est pas garanti par les compilateurs actuels.

```c
void prompt_and_print2() {
  int granted = 0;
  char buffer[10];
  gets(buffer);
  printf("%s\n", buffer);

  if (granted) {
    privileged_function();
  }
}
```

Listing 2: Fonction affichant le tampon donné en entrée, avec un accès sécurisé à une autre fonction

1. la fonction gets vient de la bibliothèque standard du C, elle lit les données sur l'entrée standard et les recopie dans le buffer passé en paramètre. Sa spécification précise que la fonction ne vérifie pas que la taille du buffer de destination est suffisante.
2. On suppose ici qu'un `int` est encodé sur 32 bits.

Il est même possible de détourner le flot de contrôle du programme de manière plus radicale. En effet, il est possible de modifier la valeur de l'adresse de retour de la fonction `prompt_and_print2` de manière à forcer le processus à retourner à l'adresse souhaitée par l'utilisateur. Un exemple serait de remplacer cette adresse de retour par l'adresse du tampon afin d'en exécuter le contenu. L'utilisateur pourrait remplir le tampon avec du code malveillant, toujours au moyen de la fonction `gets`, et le faire exécuter par le processus. De cette manière il serait possible, par exemple, d'ouvrir un shell ou de lire un emplacement mémoire arbitraire du processus.

Cette attaque consistant à modifier le contenu de la pile pour modifier le comportement du processus s'appelle le *stack smashing* (One 1996). Elle n'est cependant pas le seul type d'attaque possible grâce à la corruption de la mémoire. Szekeres et al. 2013 définit un modèle décrivant les étapes successives nécessaires au déroulement des différents types d'attaques par corruption de mémoire, ainsi que les politiques de sécurité qui permettraient de s'en prémunir. Ce modèle présente quatre grandes familles d'attaques : les attaques par corruption de code, les attaques par détournement du flot de contrôle, les attaques sur les données seulement et les fuites d'informations. Ces attaques n'ont pas toutes le même objectif, par exemple les fuites d'informations ne visent pas à induire un comportement arbitraire du processus, mais à provoquer des fuites de données sensibles alors qu'une attaque par corruption de code cherche précisément à faire exécuter du code arbitraire au processus. De plus, elles n'utilisent pas les mêmes moyens pour corrompre le processus, ainsi toutes les protections développées ne permettent pas de contrer toutes les attaques.

1.1.1 Les attaques par corruption de code

Les attaques par corruption de code consistent en l'ajout de nouveau code dans le processus ou la modification du code existant afin de modifier le comportement du processus cible. Ce type d'attaque permet d'obtenir exactement le code désiré par l'attaquant pour que celui-ci soit exécuté. Parmi les attaques de cette catégorie, on retrouve l'injection de *shellcode* exécuté grâce à la corruption d'un pointeur à l'instar du *stack smashing*, présenté précédemment.

Ces attaques sont particulièrement puissantes puisqu'elles permettent à l'attaquant d'exécuter du code arbitraire. De plus, elles sont relativement simples à mettre en place puisque l'attaquant écrit le code malveillant lui-même. Néanmoins, des protections ont rapidement été mises en place pour lutter contre ce type d'attaques. La politique de

sécurité *intégrité du code* telle que définie par Szekeres et al. 2013 a été implémentée sous la forme du W⊕X. Cette politique de sécurité considère qu'une page mémoire peut être soit inscriptible soit exécutable, mais jamais les deux en même temps. Grâce au W⊕X, les pages mémoire contenant le code du programme ne sont pas accessibles en écriture, empêchant le code du programme d'être modifié, et les pages de données accessibles en écriture ne peuvent pas être exécutées, empêchant l'attaquant d'exécuter du code qu'il aura écrit en mémoire en détournant l'utilisation normale du programme.

Bien que les attaques par injection de code aient semblé être contrées définitivement grâce à W⊕X, elles ont regagné l'intérêt des attaquants grâce à l'utilisation de plus en plus fréquente de compilateurs juste-à-temps (en anglais, *Just-In-Time* (JIT)). Ces compilateurs génèrent du code pendant l'exécution du processus, typiquement dans le but d'améliorer les performances de programmes écrits dans des langages de scripts traditionnellement interprétés (Aycock 2003 ; Chevalier-Boisvert et Feeley 2015a,b). Bien que cette technique permette d'améliorer significativement les performances et d'optimiser la compilation en fonction du contexte d'exécution, elle élargit aussi grandement la surface d'attaque du programme qui en fait usage. Par définition, afin de permettre l'utilisation de la compilation juste-à-temps, il est nécessaire d'assouplir la politique W⊕X, sinon il ne serait jamais possible d'exécuter le code généré par le compilateur. En effet, l'espace mémoire dans lequel le code compilé est écrit par le compilateur doit être accessible en écriture et exécutable ou bien rendu exécutable ultérieurement. Cela permet ainsi à un attaquant de modifier le contenu de cet espace mémoire durant la phase de compilation juste-à-temps pour injecter le code qu'il souhaite exécuter (Gawlik et Holz 2018).

1.1.2 Les attaques sur les données

Les attaques par injection de code ne sont pas toujours possibles comme présenté précédemment. De plus, exécuter du code arbitraire n'est pas nécessairement la volonté de l'attaquant, il peut se contenter de détourner le processus sans modifier son code mais en modifiant ses données. L'attaquant peut notamment modifier la valeur d'une variable pour accéder à une fonctionnalité du programme cible. Par exemple, dans le listing 3, la variable `id.type` donne accès à une interface d'administration :

```
1 void show_user_profile(identifier_t id) {
2   if (id.type == ADMIN)
3     show_admin_panel(id);
4   else
5     show_normal_profile(id);
6 }
```

Listing 3: Fonction affichant un profil utilisateur avec ou sans panel administrateur

On suppose que cette fonction permet d'afficher une page de profil pour l'utilisateur dont l'identifiant est donné en paramètre. En fonction de si l'utilisateur est un administrateur ou non, elle affiche une page d'administration ou une page de profil classique. Un attaquant possédant un compte utilisateur non privilégié sur le logiciel en question pourrait tenter de modifier la variable `id.type` correspondant à son compte afin de faire afficher une page d'administration au lieu de la page de profil attendue.

Dans ce type de scénarios, illustré par l'exemple précédent, le code du programme n'est pas impacté. Les failles exploitées servent à modifier les données d'une manière non prévue par le programmeur afin d'induire le processus en erreur. Cela conduit à violer un invariant sur les données du programme tout en suivant un flot de contrôle défini par le programme. Dans l'exemple précédent, un mot de passe d'un utilisateur qui n'est pas administrateur ne devrait pas conduire à une valeur `ADMIN` pour la variable `id.type`.

Ce type d'attaques est la base du *Data-Oriented Programming* (DOP) présenté par Hu et al. 2016, dont le principe est de corrompre les données du processus tout en suivant son flot de contrôle. Les auteurs montrent que ce type d'attaque permet en réalité de d'influencer significativement le comportement de l'application vulnérable. Pour ce faire, l'attaquant utilise des gadgets de données qui sont des petits bouts de code dont le but est de manipuler des données qui sont sous le contrôle de l'attaquant ainsi que des *gadget dispatchers* dont le travail est de chaîner les gadgets de manière arbitraire dans l'objectif de faire exécuter le code d'exploitation voulu par l'attaquant par le processus attaqué. Les gadget dispatchers étant la plupart du temps des portions de code qui bouclent, il est possible de mener ces attaques soit d'un seul coup en corrompant des données via une seule entrée soit de manière interactive en entrant de nouvelles données malveillantes à chaque itération de la boucle. Une attaque interactive peut permettre à l'attaquant de provoquer des fuites informations, qui peuvent être ré-utilisées dans la suite de l'attaque, ou d'effectuer une attaque nécessitant des corruptions de mémoire à chaque itération.

Ces attaques sont néanmoins particulièrement difficiles à mettre en place manuellement. En effet, elles nécessitent de trouver dans le code des gadgets utilisables ainsi que des gadget dispatchers capables de fonctionner ensemble, c'est-à-dire que les dispatchers doivent pouvoir diriger le flot de contrôle vers les bons gadgets sans dévier du *Control-Flow Graph* (CFG) du programme [3]. De plus, il est évidemment nécessaire de trouver une faille qui permette d'exploiter le programme cible avec une telle attaque.

Néanmoins, Ispoglou et al. 2018 ont mis au point une approche et un compilateur leur permettant d'automatiser les attaques sur les données. Pour ce faire, le compilateur utilise une trace d'exécution d'un programme vulnérable ainsi qu'un programme d'exploitation de la faille écrit dans le *Domain Specific Language* (DSL) du compilateur, *SPloit Language* (SPL). À partir de ces entrées, le compilateur est capable de générer une trace codée sous la forme d'écritures mémoires qui correspondent au code SPL donné en entrée. Cette approche simplifie grandement la mise en place de ce genre d'attaques puisqu'elle automatise la découverte de vulnérabilités exploitables là où une recherche manuelle aurait pu être infaisable.

Castro, Costa et Harris 2006 définit une implémentation de la politique de sécurité *Data-Flow Integrity* (DFI) dont l'objectif est de protéger l'intégrité des données et donc d'éviter d'utiliser des données corrompues pour détourner l'exécution du programme. L'approche proposée consiste à associer à chaque utilisation (lecture) d'un emplacement mémoire les définitions (écritures) correspondantes, de manière à définir un *Data-Flow Graph* (DFG) grâce à des analyses statiques. Le programme est également instrumenté afin de s'assurer à l'exécution que le processus définit et utilise des données conformes au DFG déterminé précédemment.

Néanmoins, cette politique de sécurité n'est pas parfaite. Typiquement, son impact sur les performances empêche son adoption à grande échelle (Castro, Costa et Harris 2006 ; Szekeres et al. 2013). De plus, l'efficacité du DFI dépend grandement de la qualité du DFG que l'implémentation de la politique de sécurité est capable de construire. Moins le DFG est précis plus il laisse de possibilité à un attaquant de trouver un moyen de le détourner. C'est autour de cette faiblesse de DFI que Lu et Wang 2019 ont développé *Data-Flow Bending* (DFB). Il s'agit d'une attaque qui cherche à construire une attaque sur les données qui respecte le DFG du programme, lui permettant de ne pas être détectée par une approche de type DFI.

3. À noter que le CFG n'est pas forcément disponible et doit alors être reconstruit de manière ad-hoc

1.1.3 Les fuites d'informations

Les fuites d'informations n'ont pas pour objectif d'atteindre à l'intégrité des données mais à leur confidentialité. Ces fuites d'informations peuvent avoir plusieurs objectifs. Elles peuvent être le but final d'une attaque, typiquement un attaquant peut chercher à obtenir le contenu d'une base de données, des secrets industriels ou encore des clefs cryptographiques. Elles peuvent aussi servir à préparer une autre attaque. Par exemple, les attaques par détournement du flot de contrôle, présentées à la sous-section suivante, nécessitent la plupart du temps de connaître l'emplacement du code en mémoire. Or, l'Address Space Layout Randomization *(ASLR)* permet de rendre l'emplacement du code aléatoire et donc rendre ces attaques plus compliquées à mettre en place. Toutefois, une fuite d'information peut permettre, par exemple, de localiser une fonction en mémoire et donc informer l'attaquant de l'organisation d'une partie (ou de la totalité) du code, lui permettant de déployer la deuxième partie de son attaque.

Bien que les attaques permettant de faire exécuter du code arbitraire au processus cible peuvent conduire à des fuites d'information, une fuite d'information ne nécessite pas forcément d'interférer avec le flot de contrôle ou le flot de données de manière si intrusive. Il est possible par exemple d'abuser de fonctions comme `printf`. En effet, si le format donné à cette fonction n'est pas une chaîne de caractères constante mais contrôlée par l'utilisateur, ce dernier peut modifier ce format pour forcer le processus à afficher n'importe quelle donnée dans l'espace d'adressage. Il est par ailleurs même possible de lui faire exécuter du code arbitraire par ce biais (Payer et Gross 2013). Il est aussi possible d'utiliser des moyens bien plus simples, comme pour l'attaque *Heartbleed* sur OpenSSL 1.0.1 (Carvalho et al. 2014).

L'attaque *Heartbleed* repose sur une faille dans le code de la fonctionnalité *hearbeat* de OpenSSL. Cette fonctionnalité implémente le mécanisme suivant : un client envoie un message ainsi que la longueur de ce message au serveur, puis le serveur renvoie une copie conforme du message au client afin de lui assurer qu'il est toujours connecté. La vulnérabilité dans OpenSSL 1.0.1 était que la taille du message n'était pas vérifiée et il était possible d'envoyer un message de petite taille tout en précisant une taille bien plus grande dans le champ du message. Le serveur renvoyait alors un message de la taille spécifiée par le client, commençant par le message d'origine puis suivi par ce qui se trouvait juste après en mémoire, possiblement des données très sensibles. OpenSSL étant une bibliothèque de cryptographie, il est donc raisonnable de penser que les données manipulées par cette bibliothèque sont très sensibles, comme par exemple des clefs de

chiffrement.

Les seules politiques de sécurités décrites par Szekeres et al. 2013 permettant de se prémunir efficacement des fuites d'informations sont la Memory Safety et une *Data-Space Randomization* (DSR) complète. La Memory Safety est une politique de sécurité qui interdit la corruption de pointeurs. Elle prévoit l'impossibilité pour un pointeur de sortir des bornes qui lui sont imposées afin de garantir l'intégrité spaciale de la mémoire. De plus, elle prévoit qu'un pointeur ne puisse pas être utilisé avant d'être initialisé ou après avoir été libéré, ce qui garantit une intégrité temporelle de la mémoire. La DSR quant à elle est une politique de sécurité qui s'assure du chargement des données manipulées par le processus à des emplacements aléatoires, empêchant un attaquant de connaître à l'avance les adresses à viser. Néanmoins, les implémentations de ces politiques de sécurité sont particulièrement coûteuses à mettre en place et imposent un surcoût en performances non négligeable (Szekeres et al. 2013).

1.1.4 Les attaques par détournement du flot de contrôle

Comme expliqué au début de section 1.1, il est particulièrement difficile aujourd'hui d'injecter du code au sein d'un processus sans le concours d'un compilateur JIT. Une autre manière de faire exécuter du code de manière arbitraire consiste à détourner le processus de son flot de contrôle. Pour ce faire, l'attaquant cherche à modifier des pointeurs de code de manière à forcer un branchement indirect à sauter à une adresse non prévue par le programmeur. Cela permet donc à l'attaquant de prendre le contrôle de l'exécution du processus.

Ces attaques peuvent prendre plusieurs formes. L'une des plus anciennes et des plus simples est l'attaque return-to-libc. Cette attaque consiste à détourner un retour de fonction de manière à sauter au début d'une autre fonction de la bibliothèque C. Cette attaque permet par exemple d'ouvrir un shell avec les privilèges du processus cible. Sur l'architecture x86 32-bits, la convention d'appels de fonctions veut que les paramètres d'une fonction soit placés sur la pile avant d'effectuer l'appel en lui-même. Ainsi, via un dépassement de tampon, il est possible de déclencher une attaque return-to-libc avec des paramètres entièrement contrôlés par l'utilisateur. La figure 1.1 détaille le contenu de la pile juste avant l'exécution d'une instruction `ret` pour un exemple d'attaque return-to-libc visant à forcer l'ouverture d'un shell.

Dans cet exemple, l'attaquant souhaite appeler la fonction `system` de la bibliothèque C avec pour paramètre la chaîne de caractères `"/bin/bash"`, dans l'objectif d'ouvrir un shell

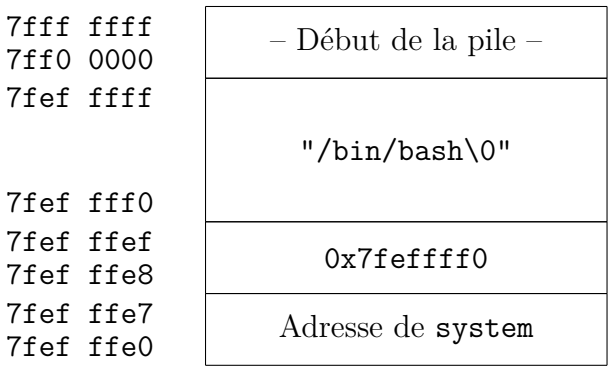

FIGURE 1.1 – État de la pile juste avant le déclenchement d'une attaque return-to-libc

Unix avec les mêmes privilèges que le processus attaqué. Pour ce faire, il écrit directement la chaîne de caractères dans la pile puis écrit l'adresse à laquelle elle commence sur la pile suivi enfin par l'adresse de la fonction `system` en mémoire. Lorsque la machine exécute l'instruction `ret`, le processus saute alors à l'adresse indiquée au sommet de la pile, celle de `system`. Son seul paramètre est une chaîne de caractère dont le pointeur se trouve au sommet de la pile, une fois le retour effectué. À partir de ce moment, le processus exécute le code de la fonction `system` et le shell est ouvert, comme le souhaite l'attaquant.

Afin de mener à bien cette attaque, aucun fragment de code n'a été injecté par l'attaquant. Aussi, la pile n'a pas besoin d'être exécutable pour que l'attaque réussisse et des protections telles que W⊕X sont inefficaces contre ce type d'attaque. Toutefois, return-to-libc n'est pas la seule attaque qui fonctionne de cette manière. Les attaques par détournement de flot de contrôle sont légions (BLETSCH et al. 2011 ; CHECKOWAY et al. 2010 ; PRANDINI et RAMILLI 2012). Il s'agit d'une des méthodes d'attaque les plus prisées aujourd'hui, non seulement parce qu'elles sont difficiles à contrer (CARLINI et al. 2015 ; EVANS et al. 2015), mais aussi parce qu'elles sont bien plus simples à mettre en place qu'une attaque sur les données telle que celles discutées Sous-section 1.1.2.

Une des attaques par détournement du flot de contrôle les plus répandues est le *Return-Oriented Programming* (ROP) (PRANDINI et RAMILLI 2012 ; SHACHAM 2007). Cette attaque consiste à exploiter le fonctionnement de l'instruction de retour de fonction (`ret` en x86) afin de faire exécuter du code arbitraire par le processus. En effet, en x86, cette instruction récupère l'adresse de retour de la fonction sur la pile et saute à cette adresse. Il est alors possible de pousser le processus à sauter à une adresse arbitraire en corrompant le contenu de la pile, tout comme avec l'attaque return-to-libc. L'idée derrière le ROP

est de sauter vers du code séquentiel, appartenant au code du processus ou d'une de ses bibliothèque partagées, qui mènera lui-même vers une nouvelle instruction `ret`. Ces bouts de code sont appelés des gadgets. Ainsi, en empilant les adresses de gadgets sur la pile, il est possible de faire exécuter arbitrairement du code par le processus cible.

Par exemple, un attaquant peut prendre le contrôle de la pile et vouloir s'en servir pour lancer un shell. Une possibilité est d'utiliser l'appel système `execve` dont l'interface en x86-64 est la suivante :

- `rdi` contient le chemin vers le fichier exécutable à lancer (un pointeur de `char`) ;
- `rsi` contient les arguments à donner au programme (un pointeur de pointeur de `char`, terminé par un pointeur nul) ;
- `rdx` contient l'environnement à donner au programme (un pointeur de pointeur de `char`, terminé par un pointeur nul).

En x86-64, un appel système est effectué en utilisant l'instruction `syscall` avec l'indentifiant de l'appel système stocké dans `rax`. Dans le cas de `execve`, il s'agit de 59. Afin de mener à bien cette attaque, il est possible de stocker le chemin du fichier executable, ici `/bin/sh`, directement sur la pile et de le faire suivre d'un pointeur nul. Ainsi, il est possible d'utiliser l'adresse de cette chaîne de caractères pour `rdi` et `rsi`. Nous n'avons pas besoin de variables d'environnement, aussi `rdx` peut être mis à nul. Sémantiquement, le code de l'attaque pourrait s'écrire ainsi :

```
1 mov rdi, _shell
2 mov rsi, _shell
3 mov rdx, 0
4 mov rax, 59
5 syscall
```

Listing 4: Appel à `execve("/bin/sh", {"/bin/sh", 0}, 0)`

Néanmoins, il serait peu probable de trouver un gadget ressemblant exactement à au fragment de code du listing 4 dans l'espace d'adressage du processus cible. Tout d'abord, il est peu probable de trouver des instructions qui placent exactement les valeurs souhaitées dans les bons registres. Par exemple, il est virtuellement impossible de trouver une instruction qui place exactement l'adresse de `/bin/sh` dans `rdi` et `rsi`. Il est plus intéressant d'essayer de trouver des gadgets de la forme `pop rdi ; ret` et de placer l'adresse voulue sur la pile. Dans le cas de l'instruction `mov rdx, 0`, il est probable de trouver des

gadgets correspondants et pour cet exemple nous considérerons que c'est le cas. Ainsi, le code de notre attaque ressemblerait au code suivant, sachant que la pile contient à son sommet deux occurences de l'adresse de `_shell` ainsi que l'entier 59 :

```
1 pop rdi
2 pop rsi
3 mov rdx, 0
4 pop rax
5 syscall
```

Listing 5: Shellcode utilisant la pile plutôt que des adresses fixes

Dans l'objectif de faire exécuter ce code par le processus cible, il est désormais nécessaire de découper celui-ci en gadgets plus petits qu'il est possible de trouver dans l'espace d'adressage du processus. La figure 1.2 donne un exemple de gadgets qu'il serait possible de trouver. Une fois les gadgets obtenus, la pile est écrite de manière à contenir les adresses de retours auxquelles sauter afin que les gadgets soient exécutés, ainsi que les données nécessaires au bon fonctionnement de ceux-ci. Une fois la pile préparée, l'exécution d'une instruction `ret` permet de démarrer l'exécution du code malveillant. La figure 1.2 montre également l'état de la pile juste avant l'exécution du premier retour de fonction.

Il est important de noter par ailleurs que l'encodage des instructions x86 est de taille variable. De fait, les instructions ne sont pas alignées et il n'est pas possible de prédire de manière fiable quelles adresses peuvent correspondre à des instructions valides dans le programme. Par conséquent, un branchement peut sauter à n'importe quelle adresse du code, y compris en plein milieu d'une instruction sur plusieurs octets, ce qui permet de décoder une instruction qui n'était pas prévue dans le code initial. De ce fait, cette particularité du code x86 augmente significativement la surface d'attaque à considérer, c'est-à-dire chaque octet qui compose le code et non plus chaque instruction.

1.2 L'intégrité du flot de contrôle

Afin de lutter contre les attaques visant à faire dévier le processus cible du flot de contrôle du programme dans le but de lui faire exécuter du code arbitraire, Abadi et al. 2005 ont mis au point une politique de sécurité permettant de certifier les branchements indirects pris par le processus. Cette politique s'appelle l'intégrité du flot de contrôle

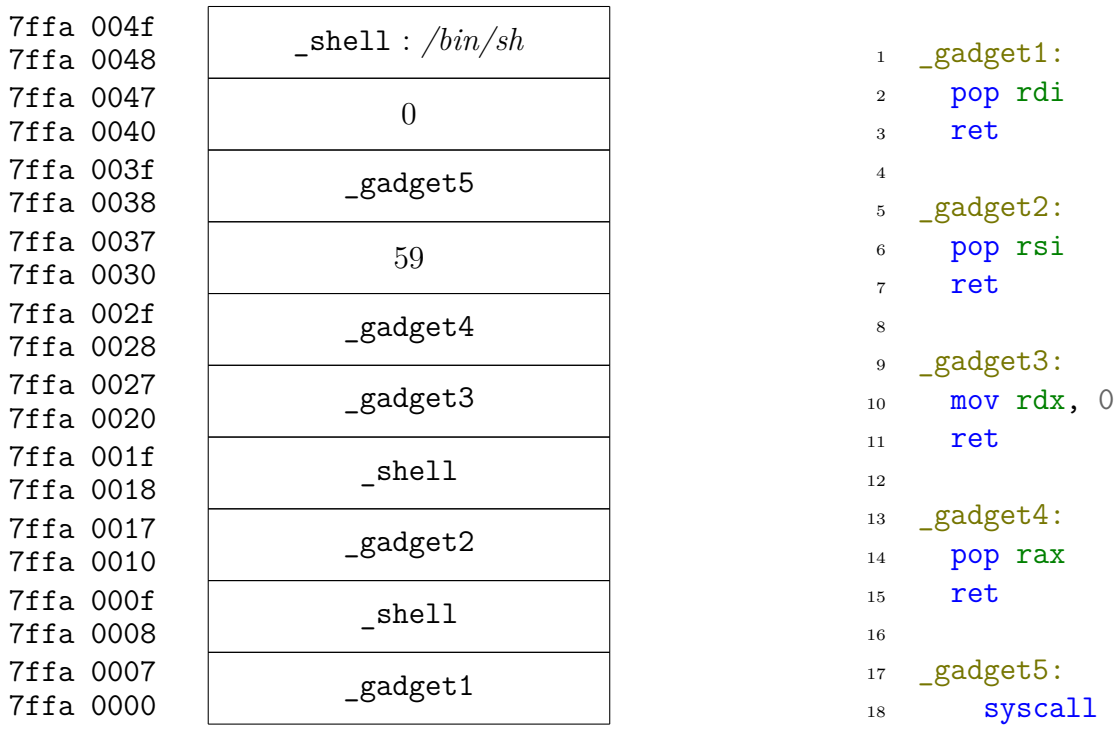

FIGURE 1.2 – Mise en place d'une attaque ROP

(*control-flow integrity*, abrégée en CFI).

Dans cet article, les auteurs décrivent précisément le fonctionnement de cette politique de sécurité. L'idée est de rassembler les branchements indirects et leurs cibles en classes d'équivalence et de leur affecter un identifiant unique. De cette manière, cette approche cherche à réduire les cibles possibles d'un saut à un sous-ensemble d'adresses qui correspond à des cibles valides définies par le flot de contrôle du programme. Lorsqu'une de ces instructions de branchement est exécutée, l'identifiant du branchement et celui de l'adresse cible sont comparés afin de s'assurer de la validité du branchement. L'intérêt de cette approche est de détecter, lors de l'exécution d'une instruction de branchement, qu'un attaquant a pris le contrôle de l'opérande de l'instruction de branchement.

Afin de mettre en place cette protection, le graphe de flot de contrôle, CFG, est utilisé. En effet, ce graphe contient toutes les transitions possibles d'un bloc de base à l'autre du programme et n'en contient aucune superflue : il est complet et minimal. Pour chaque branchement indirect du programme, toutes ses cibles valides sont connues. Pour tout point d'entrée d'un bloc de base du programme, toutes ses origines sont connues également.

L'approche proposée par Abadi et al. 2005 reconstruit le CFG depuis le binaire grâce aux informations de relocation disponibles dans les fichiers exécutables de Windows (cette approche n'est donc pas forcément portable sur d'autres systèmes). Toutefois, contrairement à un CFG construit à la compilation, il n'est pas toujours possible de s'assurer que le CFG reconstruit depuis le binaire exécutable est complet et minimal.

Grâce à aux informations obtenues depuis le CFG, il est alors possible de construire des classes d'équivalence de branchements indirects. Une classe d'équivalence contient toutes les instructions de branchements et leurs cibles telles que les branchements ont pour cibles potentielles toutes les cibles de la classe d'équivalence et aucune autre et les adresses cibles ont pour origines tous les branchements de la classe d'équivalence et aucune autre.

Chaque classe d'équivalence se voit attribuer un identifiant unique sur 32 bits, qui ne puisse pas correspondre à l'encodage d'une instruction valide. L'intérêt de cette dernière contrainte est d'éviter de créer du code supplémentaire à exploiter pour l'attaquant. Le code des instructions de branchement ainsi que celui aux adresses cibles sont instrumentés afin d'y ajouter une comparaison d'identifiants, tel qu'illustré par le listing 6.

```
1  call rdx
2
3  ; Peut être instrumenté en :
4
5  mov rax, rdx
6  cmp [rax+4], 0x12345678 ; comparaison avec l'ID
7  jne error_label
8  call rax
9  prefetchnta [0x87654321] ; ID pour la classe d'équivalence du retour.
```

Listing 6: Mise en place de CFI selon Abadi et al. 2005.

L'article de Abadi et al. 2005 suggère d'utiliser l'instruction `prefetchnta` avec l'identifiant comme opérande afin de ne pas injecter de donnée dans le code et nécessiter de sauter par dessus ces identifiants. L'instruction `prefetchnta` sert à forcer le *prefetcher* à placer l'adresse donnée en opérande dans le cache. Néanmoins, tous les processeurs n'implémentent pas cette instruction, la rendant équivalent à un `nop`. Ainsi, dans le code d'exemple précédent, lorsque la fonction appelée retourne, elle peut exécuter l'instruction `prefetchnta` contenant l'identifiant sans risque d'exécuter du code invalide ni avoir be-

soin d'un saut supplémentaire ou encore de modifier l'adresse de retour de la fonction pour éviter l'identifiant.

Cette approche est néanmoins loin d'être parfaite. En effet, l'impact de CFI sur les performances du programme cible n'est pas négligeable, sur certains programmes de la suite de benchmark SPEC2000, le temps d'exécution mesuré avec CFI majore le temps d'exécution mesuré sans CFI de plus de 30%.

De plus, cette approche permet difficilement de protéger les retours de fonction. En effet, certaines fonctions d'un programme sont susceptibles d'être appelées à de nombreux endroits dans le code et les retours de fonctions étant indirects par nature, ces instructions sont vouées à faire partie de classes d'équivalence de grandes tailles. Par conséquent, la surface d'attaque laissée à l'attaquant n'est pas négligeable. Il est intéressant d'ajouter une protection telle qu'une *shadow stack* permettant de s'assurer que l'adresse à laquelle une instruction de retour de fonction s'apprête à sauter correspond bien à l'adresse attendue. Malheureusement, l'ajout de cette protection supplémentaire a un coût en terme de performances. L'article de Abadi et al. 2005 mesure un surcoût en performances moyen de 21% mais dont plusieurs programmes dépassent les 40% pour une protection alliant un CFI et une *shadow stack*.

CFI est désormais la pierre angulaire de nombreuses recherches en sécurité applicative. En effet, de nombreuses défenses ont été mises au point depuis 2005 afin d'améliorer cette politique de sécurité sur ces trois critères : la sécurité, les performances et la disponibilité du mécanisme (c'est-à-dire sans avoir besoin des sources, des symboles, etc). Nous détaillons ces contributions dans les sections suivantes.

1.2.1 CFI basés sur les sources

CFI est très dépendant de la précision du CFG utilisé pour construire ses classes d'équivalence. Autrement dit, plus le CFG est précis, plus robuste sera la protection mise en place. Dans l'objectif d'obtenir des CFG précis, de nombreuses approches prennent la forme d'une passe de compilation plutôt que de modifier un binaire pre-existant.

En effet, le compilateur est en mesure de donner beaucoup d'information sur la sémantique du code donné en entrée et donc d'améliorer grandement la précision du CFI. Par exemple, Tice et al. 2014 propose deux mécanismes de protections implémentés sous la forme de passes de compilation dans GCC et LLVM. Ces mécanismes sont Virtual-Table Verification *(VTV)* et Indirect Function-Call Checks *(IFCC)*.

Le premier sert à protéger les appels à des méthodes virtuelles dans des programmes

écrits dans des langages objets comme C++. Les appels virtuels représentant près de 98% des appels de fonctions indirects selon Tice et al. 2014, VTV prend le pari de ne protéger que ces appels afin de réduire l'impact de la protection sur les performances. Le surcoût en temps d'exécution mesuré varie entre 0.2% et 8.7%. Le second mécanisme, IFCC, transforme les appels indirects en appels vers une table de saut. Les deux protections utilisent un fonctionnement similaire se basant sur l'ajout de métadonnées accessibles uniquement en lecture aux modules de compilation pour representer certains aspects du CFG statique du programme, ce qui permet d'offrir des informations de contexte utilisables lors de l'exécution du programme.

Un problème auquel les CFI basés sur les sources font face est celui de la compilation séparée. En effet, pour mettre en place un CFI, il est important de connaître l'intégralité du flot de contrôle du programme à protéger, or cela force le compilateur à procéder à une compilation intégrale du programme en une seule fois. En effet, lors de la compilation séparée de ses composants (bibliothèques, différents modules, etc), le flot de contrôle n'est pas complet et il n'est donc pas possible de produire un CFI complet dans ces conditions. Certaines approches comme Niu et Tan 2014a se sont intéressées à cette problématique et ont produit des approches capables de compiler séparément les différents modules d'un programme protégé par un CFI. Pour ce faire, Modular Control-Flow Integrity *(MCFI)* ajoute le type des fonctions et des pointeurs de fonction dans les métadonnées du fichier binaire en sortie du compilateur. Cette approche permet par la suite à un appel indirect de ne cibler que des fonctions dont le prototype correspond au type inscrit. Comme il est raisonnable de penser que les branchements entre deux modules différents du programme sont tous des appels et des retours de fonction (et non des sauts inconditionnels), cette approche peut générer des CFG assez précis, selon Niu et Tan 2014a.

1.2.2 CFI basés sur le code binaire

L'utilisation des informations de compilation pour déployer un CFI nécessite par définition de recompiler le programme à protéger. Toutefois, ce n'est pas toujours une option. Par exemple, un utilisateur peut avoir l'intention de protéger un logiciel propriétaire avec un CFI. Ainsi, à l'image de l'approche originelle de CFI, il existe plusieurs approches n'utilisant le fichier binaire à modifier et faisant fi du code source.

La principale difficulté lors de la mise en place d'un CFI à partir du binaire est la reconstruction du CFG. Cette étape nécessite un désassemblage complet et correct du programme, ce qui est impossible, dans le cas général, sans la présence de métadonnées

dans le fichier binaire (car analogue au problème de l'arrêt de Turing). Les approches de CFI basées sur le binaire se caractérisent donc par leur approche de désassemblage du code binaire, par la manière dont ils reconstruisent un CFG le plus précis possible ainsi que par leur façon de traiter les éventuels faux positifs détectés par leur implémentation, du fait de l'imperfection de leur CFG.

L'approche d'origine de Abadi et al. 2005 utilise les informations de relocalisation du format executable PE de Windows. Néanmoins, ces informations sont absentes des binaires au format ELF utilisés sous Linux. Les approches portables ne peuvent donc pas utiliser ces informations pour désassembler le programme à protéger.

L'article de Zhang et Sekar 2013 décrit plusieurs approches de CFI pour des exécutables binaires ne contenant pas d'information de mise au point (*stripped binaries*). Ces approches font peu d'hypothèses vis-à-vis du compilateur utilisé pour générer le binaire, telles que les conventions utilisées pour générer les tables de saut, la structure du code en mémoire, etc. Le désassemblage utilisé par cette approche est complexe. Tout d'abord, le binaire est désassemblé linéairement puis il est vérifié afin de trouver des erreurs de désassemblage et les corriger. L'étape de détection d'erreurs se base sur les vérifications suivantes : opcodes invalides, transferts de contrôle en dehors du module, transferts de contrôle au milieu d'une instruction. Une fois le code désassemblé grâce à cette méthode, le CFG est reconstruit grâce à des analyses statiques sur le code assembleur. L'intérêt de ces analyses est de déterminer certaines classes de valeurs manipulées par le programme : les constantes représentant des pointeurs vers du code, les adresses de code calculées, les adresses de gestionnaires d'exceptions, les adresses de symboles exportés ainsi que les adresses de retour de fonctions. L'article propose alors plusieurs approches de CFI utilisant les données extraites grâce à ces analyses.

De plus, cet article introduit une métrique permettant de calculer la précision d'un CFG reconstruit depuis le binaire, l'Average Indirect-target Reduction *(AIR)*, qui quantifie la fraction de potentielles cibles de branchements indirects éliminés par le CFI évalué. En d'autres termes, un CFI sans faux positifs sera d'autant plus précis, et donc sécurisant, que son AIR approche de 100%. Dans le cas de Zhang et Sekar 2013, leur approche de CFI nécessitant le moins d'informations sur le binaire à instrumenter, binCFI, atteint un AIR de 98.86% en moyenne.

Un reproche couramment fait aux CFI est leur absence de prise en compte du contexte d'exécution. En effet, les CFI s'assurent que l'exécution du programme suive bien le CFG indiqué, toutefois, il est possible d'abuser un CFI tout en restant au sein du CFG,

comme le montre Carlini et al. 2015. Par conséquent, l'utilisation d'informations concernant le contexte d'exécution pourrait permettre à un CFI de raffiner encore davantage le nombre de cibles potentielles d'un branchement indirect en fonction de l'état du processus. C'est l'approche suivie par Van der Veen et al. 2015 avec PathArmor, un CFI utilisant des fonctionnalités du matériel dans le but de tracer les chemins menant à des états sensibles du programme, permettant au CFI de définir des transitions valides dans le CFG selon le contexte d'exécution.

1.2.3 L'isolation des données de contrôle

L'intégrité du flot de contrôle est une politique de sécurité qui vise à garantir que le processus ne dévie pas du flot de contrôle voulu par le programmeur. Pour ce faire, il ajoute des vérifications à l'exécution permettant de s'assurer que les branchements indirects ciblent des adresses valides et non du code arbitraire. Cette approche a néanmoins deux principaux défauts. Tout d'abord elle rajoute du code à exécuter par le processus cible, ce qui a pour conséquence de le ralentir dans sa tâche. Ensuite, elle ne supprime pas la cause principale des déviations du flot de contrôle, à savoir la présence des instructions de branchements indirects.

Control-Data Isolation (CDI) (Arthur et al. 2015) décrit une nouvelle approche consistant à supprimer ces branchements indirects afin de se débarrasser du problème plutôt que de rajouter des gardes-fous additionnels. Des tobbogans sont écrits à la place des branchements indirects dans l'objectif de les remplacer par des branchements directs.

Les tobbogans

Arthur et al. 2015 définissent une structure de contrôle qu'ils appellent un *sled* (traduit ici en tobbogans). Il s'agit d'une séquence de pairs de sauts conditionnels et de branchements inconditionnels directs permettant de remplacer un branchement indirect. L'idée est de comparer l'adresse cible du branchement indirect avec des valeurs immédiates écrites dans le code. Ces comparaisons permettent de trouver une entrée correspondante à l'adresse cible. Cette entrée réalise ensuite un branchement direct à l'adresse cible à la place du branchement indirect. Les tobbogans sont décrits plus en détails dans la sous-section 2.3.1.

CDI est implémenté sous la forme d'une passe de compilation au sein de LLVM. Il utilise le CFG généré par le compilateur afin de transformer, dans un premier temps, chaque branchement indirect en une structure appelée branchement à cibles multiples (*multi-way branchment*, abrégés en MBR dans l'article originel). Ces MBR sont par la suite compilés en des tobbogans d'instructions natives, donnant des codes similaires au code suivant :

```
1  ;; On suppose ici que rax
2  ;; peut cibler :
3  ;;    - func1
4  ;;    - func2
5  ;;    - func3
6  call rax
```

```
1  _case_func1:
2    cmp rax, func1
3    jne _case_func2
4    call func1
5    jmp _end_sled
6  _case_func2:
7    cmp rax, func2
8    jne _case_func3
9    call func2
10   jmp _end_sled
11 _case_func3:
12   cmp rax, func3
13   jne _case_error
14   call func3
15   jmp _end_sled
16 _case_error:
17   call _error_handler
18 _end_sled:
```

Bien que cette solution permette d'éviter qu'un processus échappe complètement au CFG prévu, elle reste néanmoins vulnérable à des attaques telles que le control jujutsu (Evans et al. 2015) ou le control-flow bending (Carlini et al. 2015). En effet, bien que cette approche limite les cibles possibles d'un branchement à celles prévues dans le code, elle n'est pas sensible au contexte d'exécution. Ainsi, si un attaquant modifie l'adresse cible d'un branchement de manière à rediriger l'exécution vers une autre cible valide, le comportement du programme est changé sans que la protection mise en place ne puisse le détecter.

L'approche proposée par Arthur et al. 2015 pèche par ailleurs par son impact sur les performances du programme cible en l'absence d'optimisations. En effet, l'implémentation

naïve de CDI impose un surcoût moyen de 45% sur l'ensemble des benchmarks utilisés dans l'article. Dans le pire cas, PERLBMK, le temps d'exécution est presque doublé. Il est toutefois possible d'optimiser le fonctionnement de CDI en modifiant le contenu des tobbogans, comme décrit dans l'article originel. Par exemple, les branchements indirects ne pouvant cibler qu'une seule adresse peuvent être simplifiés en une unique instruction de branchement direct plutôt qu'en un tobbogan à une entrée. Les fonctions les plus appelées peuvent être clonées afin que le tobbogan de retour de fonction devienne un simple saut direct (ainsi qu'un ajustement du pointeur de pile) et que chaque endroit appelant cette fonction dans le code appelle désormais son exemplaire unique de la fonction. Cela permettrait par ailleurs d'améliorer la précision du CDI concernant les retours de fonctions puisque la fonction appelée dépend directement de l'instruction appelante (l'adresse de retour est donc unique et connue statiquement). Cette optimisation est pratique pour les petites fonctions appelées depuis de nombreux endroits mais implique un lourd sacrifice en mémoire pour les plus grosses fonctions. Il est possible aussi de trier l'ordre d'apparition des entrées d'un tobbogan pour que les plus utilisés apparaissent en premier.

Parmi les optimisations décrites dans l'article de ARTHUR et al. 2015, une optimisation propose de trier les entrées des tobbogans. Bien que des ordres complexes puissent être mis en place comme, par exemple, pour faire en sorte de faciliter le travail du prédicteur de branchement (ARTHUR et al. 2015), l'ordre le plus simple est de placer les entrées des tobbogans par ordre décroissant de fréquence d'utilisation. Malheureusement il n'est pas toujours possible de déterminer statiquement quelles entrées seront les plus visitées.

Lors de leurs expérimentations, Arthur et al. ont utilisé une compilation en deux temps. La première compilation permet de mettre en place un CDI non optimisé. Le programme généré est exécuté afin de profiler l'utilisation des entrées des tobbogans qu'il contient. Enfin, une seconde compilation du programme utilise les traces d'exécution générées précédemment et permet de générer des tobbogans optimisés.

1.3 La modification dynamique de binaire

La modification dynamique de binaire (*Dynamic Binary Modification* ou DBM) est une technique dont l'objectif est de modifier un processus durant son exécution. Elle permet de modifier les données ainsi que le code du processus. Pour ce faire, il est possible d'instrumenter le code du processus directement depuis son espace d'adressage en y injectant tout un moteur de DBM. C'est l'approche suivie par Pin (LUK et al. 2005) ou

DynamoRIO (D. Bruening, Garnett et S. Amarasinghe 2003), qui injectent dans le processus cible une bibliothèque partagée contenant le code d'instrumentation. Il est aussi possible de prendre le contrôle d'un processus cible depuis un processus extérieur, comme le font les débogueurs, et d'instrumenter le processus sans que le client de modification de binaire ne partage son espace d'adressage avec le processus cible.

Comme le détaille longuement Hazelwood 2011, la DBM a de nombreuses utilités et est destinée à de nombreux acteurs. Tout d'abord les développeurs de logiciels peuvent s'en servir afin d'analyser leurs programmes. Le framework Valgrind (Nethercote et Seward 2007) permet typiquement de détecter les fuites mémoires et d'effectuer du profilage de code. Les développeurs peuvent également utiliser la DBM dans le but de détecter et supprimer les bugs de leurs programmes, c'est en effet la technique utilisée par les débogueurs pour effectuer leur travail.

Il est possible aussi d'utiliser la DBM pour optimiser le fonctionnement du programme cible en tirant parti des informations disponibles à l'exécution. Par exemple, un programme distribué à grande échelle ne sera pas compilé avec des optimisations spécifiques, afin de permettre l'exécution sur une large gamme de processeurs sans nécessiter de recompiler le programme. Typiquement, un programme pourrait être compilé sans utiliser les extensions AVX, disponibles dans les processeurs les plus récents. Un client de DBM pourrait analyser le code du processus et modifier les instructions SSE ou AVX dans le but de les remplacer par des instructions plus récentes et donc possiblement plus efficaces (Hallou 2017 ; Hallou, Rohou et Clauss 2017). Il est possible aussi de profiler l'exécution du processus afin de spécialiser les fonctions les plus utilisées pour en améliorer les performances, comme le propose FittChooser (Ap et al. 2018).

Les développeurs peuvent aussi recourir à la DBM dans le but de mettre en place des sécurités à l'exécution du programme. C'est typiquement le cas du *program shepherding* (Kiriansky, Derek Bruening et Saman Amarasinghe 2002), une approche surveillant les transferts de flot de contrôle durant l'exécution pour s'assurer du bon fonctionnement du programme. De part le contrôle offert par la modification dynamique de binaire sur le processus instrumenté, il est possible de construire toute une *sandbox* autour de celui-ci et d'analyser son comportement dans un environnement contrôlé, ce qui est particulièrement intéressant pour de l'analyse de programmes malveillants (*malware*).

Les architectes de processeur peuvent aussi avoir recours à la DBM. Cette technique permet par exemple d'émuler une instruction encore inexistante. Le processus exécute le programme normalement jusqu'à rencontrer une nouvelle instruction introduite par

l'architecte. Le code d'instrumentation prend alors le relai afin d'imiter le comportement de l'instruction, puis rend le contrôle au processus cible. Il est aussi possible de simuler un composant tel qu'un prédicteur de branchement afin de le tester et d'étudier son comportement avant de le construire physiquement.

1.3.1 La DBM dans la sécurité

La DBM est une technique particulièrement intéressante dans le cadre de la cyber sécurité. En effet, cette technique permet des analyses poussées et précises de programmes sans avoir besoin du code source de ces derniers. Par exemple, BlackBox (Hawkins, Demsky et Taylor 2016) surveille l'exécution du programme cible afin d'en extraire des traces d'exécution et d'en déduire un profil fiable d'exécution, permettant de mettre en place des protections pour de prochaines exécutions.

Néanmoins, l'utilité de la DBM dans le cadre de la sécurité ne s'arrête pas là. Diverses approches inspirées de CFI utilisent cette technique. L'intérêt de la DBM pour ces approches est multiple. Par exemple, elle peut permettre au programme cible de fonctionner correctement en présence de protections, malgré les modifications apportées au code d'origine. C'est typiquement le cas de BinCFI (Zhang et Sekar 2013) qui utilise un chargeur dynamique modifié afin de remplir des tables de redirection à chaque chargement d'un nouveau module à l'exécution. Cette modification dynamique permet de rediriger correctement les appels indirects vers les fonctions de ce module. En effet, BinCFI traduit les branchements indirects en vérifications inspirées par CFI. Néanmoins, cette approche place le code instrumenté à un emplacement différent du code originel, modifiant ainsi toutes les adresses cibles des branchements. C'est la raison pour laquelle cette approche utilise des tables, afin de traduire chaque pointeur et rediriger le flot de contrôle vers le code instrumenté, lors de l'exécution de branchements indirects. Notre approche, présentée au Chapitre 2, utilise un système similaire et le besoin d'utiliser des tables de redirection, rencontré par BinCFI, y est décrit plus en détail à la Section 2.3.

D'autres approches telles que le *program shepherding* (Kiriansky, Derek Bruening et Saman Amarasinghe 2002) sont entièrement basées sur l'instrumentation dynamique de binaire et contrôlent complètement l'exécution du processus cible, permettant de vérifier divers invariants pendant l'exécution du programme. L'approche utilisée consiste en la copie progressive du programme dans des caches de code, au fur et à mesure de son exécution, tout en instrumentant le code avec des vérifications dans le but de faire respecter un certain nombre de politiques de sécurité.

Plus récemment, une approche de CFI complètement dynamique a été développée par Payer, Barresi et Gross 2015 sous le nom de Lockdown. Cette approche ne nécessite aucune analyse statique avant l'exécution du programme cible et s'efforce d'injecter des protections durant son exécution. Pour ce faire, Lockdown traduit tous les blocs de base du programme au fur et à mesure de l'exécution et injecte des vérifications dès lors qu'un branchement indirect apparaît. Lockdown est implémenté grâce à une bibliothèque de DBM, *libdetox*, dont la particularité est d'être spécifiquement orientée vers la sécurité. En effet, *libdetox* utilise un mécanisme de traduction de la pile et des régions mémoire séparées de l'application en cours d'exécution afin d'empêcher celle-ci d'interférer avec le code d'instrumentation et compromettre la sécurité du processus. Cette bibliothèque s'appuie sur un chargeur dynamique capable de protéger le système de *Dynamic Binary Translation* (DBT) d'attaques contre le chargeur, lors du chargement et du déchargement de bibliothèques partagées. Par conséquent, Lockdown contrôle complètement l'exécution du processus cible et n'a pas la possibilité de s'y attacher lorsque celui-ci est déjà en cours d'exécution.

1.4 Limites de l'état de l'art

Malgré leurs qualités, les approches décrites précédemment ont toutes un défaut en commun : elles nécessitent d'être appliquées avant le lancement du programme. En effet, bien que certaines utilisent des vérifications faites à l'exécution du programme, le code de ce dernier est instrumenté en amont, soit à la compilation (Arthur et al. 2015), soit ultérieurement en modifiant directement le fichier binaire (Abadi et al. 2005) ou en utilisant un chargeur dynamique personnalisé afin de lancer le processus, permettant un contrôle plus poussé du comportement du processus qu'il serait difficile d'obtenir sans l'aide extérieur du chargeur dynamique Zhang et Sekar 2013.

Le problème posé par cette limitation est qu'un processus en cours d'exécution ne peut pas être protégé a posteriori. En effet, une fois le processus en cours d'exécution, il n'est plus possible de le modifier de manière traditionnelle, en changeant ses sources, le fichier binaire ou le chargeur. Il est nécessaire alors d'arrêter l'exécution du processus et relancer son exécution par la suite. Néanmoins, il existe des processus qu'il est préférable de ne pas arrêter. Ils peuvent être soit très longs à démarrer ou à arrêter, soit leur mise à l'arrêt peut avoir des conséquences néfastes comme une indisponibilité d'un service critique. Pour ces processus, une instrumentation du code à chaud serait une solution plus intéressante.

Il serait néanmoins possible de modifier le programme afin d'y ajouter des sécurités alors que celui-ci est déjà en cours d'exécution. Un tel fonctionnement impliquerait de s'attacher au processus et d'en prendre le contrôle pour l'instrumenter. En réalité, il s'agit déjà du comportement des débogueurs tels que GDB ou LLDB. Le contrôle du processus cible est obtenu par le débogueur grâce à un mécanisme tel que `ptrace` sous Linux et son code est instrumenté afin d'y placer des mécanismes comme les *breakpoints*. Le compilateur peut également modifier temporairement le code dans le but d'appeler des fonctions dans le processus cible ou de lire des informations dans sa mémoire.

Il est possible de faire usage de ces mécanismes afin de modifier complètement le code exécuté par le processus afin d'y ajouter des mécanismes de protection. Nous avons exploré ces possibilités afin de développer une approche complètement dynamique permettant d'assurer l'intégrité du flot de contrôle.

1.4.1 Notre contribution

Durant notre travaux de doctorat, présentés dans ce manuscript, nous avons développé une approche du CFI dont la particularité est d'être mise en place durant l'exécution du programme cible. En effet, notre approche ne requiert aucune instrumentation a priori du programme, ni à la compilation, ni via une modification du fichier binaire. Notre approche ne fait pas de supposition concernant le chargeur dynamique et ne nécessite l'ajout d'aucun module noyau. Notre outil s'attache à un processus existant, tout comme le ferait un débogueur, puis en prend le contrôle dans le but de le protéger. Cette approche prend la forme d'un framework permettant l'implémentation de protections à l'exécution. Nous avons implémenté, grâce à ce framework, une protection basée sur CDI que nous avons adaptée à une mise en place lors l'exécution du programme cible.

L'implémentation de notre approche cible l'ISA x86-64 et le système d'exploitation Linux. Nous utilisons des mécanismes propres à ce dernier (`ptrace` notamment). Néanmoins il est possible d'adapter notre approche à d'autres architectures de processeurs ainsi qu'à d'autre systèmes d'exploitation, en modifiant les parties spécifiques à ces derniers.

Notre implémentation se compose principalement de trois parties :

— Sorry, une bibliothèque de DBM orientée vers les performances et l'intervention minimale sur le processus cible ;

— Damas, un framework de protection de processus à l'exécution ;

— Une implémentation dans Damas d'une approche dynamique fortement inspirée de CDI.

Nous avons mené des expérimentations sur notre outil afin de mesurer son impact sur les performances du processus cible. Nous avons mis en place des optimisations permettant d'améliorer ces performances et développé un système de mise à jour dynamique du programme instrumenté, de manière à réduire l'impact de notre approche sur le temps d'exécution du processus cible, en faisant usage des informations disponibles à l'exécution. Dans le but de nous assurer de la validité des protections mises en place ainsi que de leurs limites, nous avons mis en place un certain nombre de scénarios d'attaques. Nous avons attaqué des processus sans protection et protégés avec Damas et comparé les résultats. Ces travaux ont fait l'objet d'une publication à SILM 2021 (Le Bon et al. 2021).

Chapitre 2

Damas : Un framework d'injection de protection à l'exécution

L'objectif de notre approche est d'éviter qu'un processus diverge du flot de contrôle prévu. Contrairement au CFI, nous ne vérifions pas la validité des branchements indirects durant l'exécution du processus, nous les supprimons. En effet, les branchements indirects sont la cause principale des attaques par déviation du flot de contrôle. Les branchements directs sont les branchements dont l'opérande est un immédiat ou un décalage immédiat par rapport au pointeur d'instruction. Puisque l'opérande des branchements directs est écrite en dur dans le code du programme, il n'est pas possible de les détourner sans utiliser de moyens en dehors du cadre de nos recherches tels que les injections de fautes. À l'inverse, les branchements indirects lisent l'adresse cible du branchement dans la mémoire pendant l'exécution. Si un utilisateur parvient à modifier la valeur à l'adresse désignée par l'opérande d'un branchement, il lui est alors possible de faire dévier le programme de son flot de contrôle originel. En supprimant les branchements indirects, il est possible d'empêcher ces attaques de se produire.

Terminologie concernant les branchements

Dans ce manuscript, nous ferons la différence entre la notion générale de *branchement* qui désigne toute forme d'instruction de transfert de contrôle et les instructions de *saut* qui désignent précisément les instructions `jmp` et `jmpq` du jeu d'instructions x86_64. Le tableau 2.1 définit plus précisément la terminologie employée. Nous avons choisi ces termes pour des raisons de clarté, de cohérence et pour s'accorder au reste de la littérature qui appelle *branchement indirect* tout type de branchement dont l'opérande n'est pas un immédiat.

Nom générique	Categorie	Exemples (x86_64 ISA)
branchement	saut	`jmp [rip+0xd7f9a]` `jmp [0x4c5580+rax*8]`
	appel	`call [rax]` `call [r15+rbx*8]`
	retour	`ret`

Table 2.1 – Notre terminologie concernant les instructions de transfert de contrôle

Contrairement aux travaux précédents sur CDI, notre approche se base sur une représentation binaire du programme cible pour lequel nous n'avons pas d'informations de compilation, telles que le CFG, la signature des fonctions, etc. Nous nous intéressons en revanche à des processus en cours d'exécution. Par conséquent, nous avons un accès à diverses informations sur l'exécution du processus telles que la cartographie de la mémoire, un accès total à l'espace d'adressage du processus ou encore les valeurs des registres. Nous nous intéressons à des programmes écrits dans les langages de programmation comme le C puis compilés grâce à des compilateurs tels que GCC ou LLVM, sans obfuscation du code ni utilisation de code assembleur écrit à la main. L'idée est de pouvoir protéger un programme classique sans tenir compte des difficultés rencontrées dans du code exotique. Les hypothèses suivantes sont faites :

— le code binaire n'est pas obfusqué ;
— il n'y a pas d'assembleur écrit à la main rendant le désassemblage plus compliqué ;
— chaque fonction n'a qu'un unique point d'entrée.

Dans ce chapitre, nous présentons notre approche ainsi que son implémentation sous la forme d'un outil de protection de processus à l'exécution nommé Damas. Dans un premier temps, Section 2.1, nous présentons notre bibliothèque de modification dynamique de binaires, Sorry, et expliquons les particularités de celle-ci par rapport aux bibliothèques concurrentes. Dans un second temps, Section 2.2, nous décrivons les difficultés rencontrées lors de la modification du code d'un processus durant son exécution et expliquons la nécessité de le réécrire entièrement, justifiant un rebasement en mémoire de l'entièreté du code. Nous détaillons aussi les mécanismes mis en place afin d'assurer ce rebasement en mémoire sans interférer avec la sémantique du programme. Dans un troisième temps, Section 2.3, nous expliquons comment les sauts indirects sont supprimés du programme et par quoi ils sont remplacés afin de préserver la sémantique du programme tout en protégeant son flot de contrôle.

Rebasement en mémoire

Dans ce manuscript, il sera de multiples fois fait mention de *rebasement en mémoire*. Il s'agit d'une traduction du terme anglais *relocation* et sera utilisé dans le sens de prendre un objet (un bloc de base, une fonction, une instruction, etc) en mémoire et de le placer à une autre adresse.

2.1 Sorry : notre bibliothèque de DBM

Notre approche a pour but de protéger un processus déjà en cours d'exécution de possibles déviations de son flot de contrôle. Dans ce but, notre approche utilise la modification dynamique de binaire afin de réécrire les parties du programme cible susceptibles d'être abusées par un utilisateur mal intentionné. Il existe plusieurs outils différents permettant de modifier un processus durant son exécution. Néanmoins les principaux candidats ne respectaient pas les contraintes imposées par notre approche, à savoir :

— pouvoir se greffer sur un processus existant ;
— avoir un impact minimal sur les performances du programme cible.

En effet, nous avons considéré les outils suivants : Pin (Luk et al. 2005), DynamoRIO (D. Bruening, Garnett et S. Amarasinghe 2003) et Padrone (Riou et al. 2014). DynamoRIO ne permettait pas de s'attacher à un processus en cours d'exécution lors de notre choix de technologie [1], le disqualifiant d'office. Pin permet cela, mais son fonctionnement est plus problématique. En effet, un *pintool* instrumente complètement l'exécution du processus cible, recompilant l'intégralité de son code à la volée dans le but d'y greffer les fonctionnalités souhaitées. Bien que ce système soit intéressant et aurait grandement facilité le développement de Damas, il impose un surcoût important en temps d'exécution (Luk et al. 2005). Pour cette raison, nous avons également préféré ne pas utiliser Pin.

Padrone est une bibliothèque de DBM légère se basant essentiellement sur l'appel système `ptrace` et les compteurs de performances disponibles sur le système d'exploitation GNU/Linux. Contrairement à Pin et DynamoRIO, Padrone n'instrumente pas l'intégralité du processus cible et n'interagit avec ce dernier que pour le modifier, par

1. L'option `-attach` de `drrun` n'est apparue que durant l'année 2019, soit un an après le début de nos travaux.

exemple en injectant du code ou en redirigeant le flot de contrôle vers du code injecté précédemment. Ainsi, si l'utilisateur ne fait aucun autre traitement que s'attacher au processus cible, Padrone n'impose aucun surcoût en temps d'exécution, contrairement à Pin.

Malheureusement, la bibliothèque n'étant plus développée depuis le départ de son principal contributeur, il ne nous a pas été possible de l'utiliser pour développer notre outil. Par conséquent, nous avons développé une nouvelle bibliothèque, Sorry, basée sur les idées de Padrone. Sorry suit la même philosophie que Padrone et n'interfère que très peu avec l'exécution du processus cible, permettant de réduire au maximum son impact sur les performances de celui-ci. En outre, Sorry a été pensée dès le départ pour s'attacher à un processus existant sans nécessiter de relancer le programme cible.

Sorry est écrite en Rust afin de faciliter son développement par rapport à une bibliothèque écrite en C comme Padrone. Par ailleurs, les garanties offertes par Rust permettent de nous assurer plus facilement de la solidité du code qui compose notre bibliothèque. Ce choix est aussi motivé par l'écosystème de Rust qui permet très facilement d'importer des dépendances, de tester notre code et de le redéployer au sein d'autres projets, comme notre outil Damas.

2.1.1 Philosophie et fonctionnalités

L'un des objectifs de Sorry étant d'imposer un surcoût en performances minimal au processus qu'elle instrumente, elle a été conçue pour avoir le moins d'interactions possibles avec ce dernier, tant que ce n'est pas nécessaire et souhaité par le développeur du client Sorry. Cela signifie que le code du processus cible n'est pas instrumenté par défaut et que Sorry n'intervient pas dans l'exécution du processus par un quelconque moyen (signaux, *breakpoints*, etc.) sans que cela n'ait été explicitement demandé par l'utilisateur de Sorry. Tout comme Padrone, Sorry utilise principalement l'appel système `ptrace` pour manipuler l'exécution du processus cible.

Afin de limiter l'impact de Sorry sur les performances du processus cible, rien n'est fait automatiquement sans le concours du développeur. Typiquement, contrairement à un framework plus intrusif comme Pin, le flot de contrôle du processus cible n'est ni instrumenté, ni recompilé. Afin de prendre le contrôle de l'exécution du processus, le client Sorry est contraint d'utiliser des points d'arrêt (des *breakpoints*). Ainsi, le programme n'est perturbé par le client que lorsque c'est nécessaire. À l'inverse, il n'est pas toujours nécessaire de limiter l'impact de notre bibliothèque sur les performances du client qui

l'utilise. En effet, on pourrait imaginer des applications où le client peut effectuer des traitements lourds en parallèle de l'exécution du processus cible, sans interférer avec lui, et ne le modifie qu'au dernier moment. Par conséquent, Sorry met à disposition des outils et des APIs permettant un développement confortable de clients d'analyse et de modification de processus.

Une des principales fonctionnalités de Sorry est d'exposer une API simple qui permet de représenter un processus cible et les interactions possibles avec lui, tout en masquant les détails d'implémentation sous-jacents, tels que les appels à `ptrace`. Par ailleurs, cette représentation abstraite du processus cible permet d'accéder de manière intuitive à diverses informations pertinentes comme, par exemple, les informations du fichier ELF du programme ou encore la cartographie de la mémoire permettant de connaître les adresses de chargement des bibliothèques liées au processus.

Le code du listing 7 est un exemple de client Sorry qui se greffe à un processus dont le PID est 12345 et récupère l'adresse à laquelle la bibliothèque C est chargée en mémoire ou affiche un message d'erreur si elle n'est pas chargée :

```
1  let target = TargetProcess::from_pid(12345)?;
2  let ctrl = target.get_controller();
3  ctrl.load_map_file()?;
4
5  let option = ctrl.find_memory_map_entry(|entry| {
6    entry.permissions.is_executable() &&
7    entry.filename.unwrap_or("").contains("libc.so")
8  });
9
10 match option {
11   Some(entry) => println!("LibC loaded at: {:#x}", entry.start_addr),
12   None => println!("No libC is loaded in the target process")
13 }
```

Listing 7: Client Sorry affichant l'adresse de chargement de la bibliothèque C dans le processus cible

Ce client très simple montre bien comment Sorry fonctionne. Un objet de type `TargetProcess` est instancié grâce au PID d'un autre processus et servira à contenir

toutes les informations nécessaires à la manipulation du processus cible. La plupart des manipulations sont effectuées par la suite par un objet intermédiaire, le contrôleur, permettant de séparer l'état du processus (le `TargetProcess`) des opérations effectuées dessus. De plus, l'utilisation d'un objet séparé pour manipuler le processus cible permet l'écriture de clients complexes, utilisant plusieurs `TargetController` différents, tout en s'assurant que le `TargetProcess` reste unique. Cette architecture permet d'éviter que les informations contenues dans un `TargetProcess` ne soient incohérentes par rapport à l'état réel du processus cible.

À la ligne 3, le fichier contenant le mapping de la mémoire est chargé manuellement. Ce fichier correspond au fichier `/proc/PID/maps`. Comme expliqué précédemment, Sorry ne fait presque rien automatiquement pour éviter d'impacter lourdement les performances du processus cible. Le chargement (ou rafraîchissement) de la représentation interne du mapping de la mémoire du processus cible est un bon exemple. En principe, afin que cette représentation interne soit toujours à jour par rapport au fichier `/proc/PID/maps`, il faudrait effectuer ce rafraîchissement à chaque arrêt du processus cible. Cependant, si le processus est amené à être souvent arrêté, cela peut avoir un impact non négligeable sur la capacité du client à le relancer au plus vite. Pour cette raison, le développeur prend la responsabilité de rafraichir la représentation interne du mapping de la mémoire à chaque fois qu'il en a besoin.

Par la suite, de la ligne 5 à 8, la méthode `find_memory_map_entry` permet de trouver la première région de la mémoire du processus cible qui répond à un prédicat donné en paramètre. Cette exemple illustre par ailleurs l'utilité de la variable `entry` dont le type est `MemoryMapEntry` et permet de raisonner sur les entrées du fichier `/proc/PID/maps` simplement. Le reste de la bibliothèque est à cette image : les types de données permettent de raisonner intuitivement sur le processus cible mais tous les traitements sont effectués à la demande.

2.1.2 Lectures et écritures via les *tampons*

Une des principales fonctionnalités de Sorry est son système de *tampons*. Un client Sorry est amené à modifier ou à inspecter la mémoire de son processus cible, que ce soit son code ou ses données. Pour ce faire, Sorry propose plusieurs type de tampons qui permettent d'effectuer ces manipulations sur la mémoire du processus cible de manière simple. L'objectif de ces tampons est de masquer la complexité des opérations effectuées par la bibliothèque pour lire ou écrire dans la mémoire du processus cible tout en assurant

la validité des manipulations décrites par le développeur. Par exemple, un tampon doit s'assurer que les adresses auxquelles il doit accéder sont des adresses valides dans l'espace d'adressage du processus cible. Un tampon permet aussi de raisonner sur des décalages par rapport à une adresse de base, par exemple en raisonnant sur des décalages par rapport au début d'un tableau ou de l'adresse de début d'une fonction.

Les traits de Rust

Rust est un langage fortement typé dont le système de type permet d'écrire du code très générique. Une fonctionnalité du langage, les *traits*, permet de décrire une interface contenant des fonctions ou des types associés que les types peuvent implémenter. Implémenter un trait pour un type permet de donner à tout objet de ce type les fonctionnalités promises par le trait. L'utilisation de ces traits permet d'écrire du code très générique comme, par exemple, des fonctions ne prenant plus une variable d'un type concret en paramètre, mais une variable de tout type implémentant un trait précis.

Ces tampons prennent la forme d'un trait et de plusieurs types de données. Le trait `Buffer` définit une interface commune à tous les types considérés comme des tampons, exposant ainsi un certain nombre de méthodes permettant de manipuler des tampons de manière transparente. En effet, le système de type de Rust admet le polymorphisme paramétrique ainsi qu'un polymorphisme ad-hoc permettant de contraindre les paramètres de type utilisés. Ainsi, il est simple de définir des fonctions prenant en paramètre une variable d'un quelconque type implémentant le trait `Buffer` ou même des types de données abstraits paramétriques dont les paramètres de types sont contraints à implémenter `Buffer`. Par exemple, il serait possible de représenter une bibliothèque chargée en mémoire dans le processus cible avec un type comme celui du listing 8.

```
1 struct Library<B: Buffer> {
2   filename: String,
3   buffer: B
4 }
```

Listing 8: Définition d'un type dont le paramètre est contraint par le trait `Buffer`

Il existe deux principaux types de tampons dans Sorry : les `RemoteBuffer` et les `CodeCache`. Les `RemoteBuffer` sont des tampons ne nécessitant pas d'allocation de mémoire dans le processus cible. Leur objectif est d'ouvrir une vue sur une zone mémoire préexistante du processus cible, permettant lecture ou écriture. Les `CodeCache` sont des tampons qui allouent une zone mémoire dans l'espace d'adressage du processus cible. Ces tampons sont utiles pour injecter du code ou des données dans le processus cible. L'intérêt de ce type de données est de cacher au développeur toutes les étapes nécessaires à l'allocation du tampon.

Grâce à l'API de Sorry, la manipulation de la mémoire d'un autre processus prend la forme de manipulation de tableaux dont l'utilisation est simple et sûre. Elle permet qui plus est de raisonner de manière abstraite sur les tampons sans se soucier du type de tampon manipulé. De plus, nous avons écrit une implémentation du trait `Buffer` pour le type de données Rust représentant un tableau dynamique, `Vec`. Cela permet d'écrire facilement des tests pour des opérations manipulant des tampons. Pour cela, un `Vec` peut être utilisé à la place d'un véritable tampon, permettant de vérifier si les lectures ou écritures dans le tableau sont conformes à celles attendues au sein du tampon correspondant. De cette manière, les tests écrits ne nécessitent pas le lancement d'un processus tiers, réduisant au maximum les chances qu'un test échoue pour des raisons extérieures à la logique du test lui-même.

2.2 Control-Data Isolation à l'exécution

Dans cette section, nous entrons dans les détails de notre approche. Nous expliquons les principaux mécanismes qui définissent notre framework de protection de processus à l'exécution et utilisons notre implémentation d'une protection de type CDI en guise d'exemple d'utilisation. Cette section se découpe essentiellement en trois parties, la première partie explique le besoin de rebaser l'intégralité du code binaire du programme en mémoire, la seconde partie explique la manière dont Damas procède à ce rebasement et la troisième partie explique les différents changements à apporter au code du programme afin de conserver la sémantique de celui-ci, une fois le rebasement effectué.

2.2.1 Contraintes spatiales dans le binaire originel

La différence entre le CDI originel et notre approche dynamique est importante. Au lieu de remplacer chaque branchement avec un tobbogan raisonnablement petit, nous créons des classes d'équivalence de branchements qui sont redirigés vers un seul code plus gros à qui le transfert de contrôle est délégué. La sous-section 2.3.3 décrit les structures de code responsable du traitement des instructions de branchement. Ces structures, que nous appelons tables de délégations (*dispatch tables*) sont plus complexes que les tobbogans de CDI.

Les caves de code

Les caves de code (*code caves*) sont des petits espaces initialement inutilisés dans la mémoire qui sont mis à profit lors de l'instrumentation du code. Typiquement, lors de l'instrumentation statique de code binaire, il n'est pas toujours possible d'injecter du code en un seul bloc ou d'agrandir la section `.text` pour ajouter du code à la fin voire même de rajouter une section à la fin du fichier ELF. Modifier la disposition initiale du fichier pourrait invalider le code existant. En effet, l'ajout d'une section dans un fichier ELF nécessiterait d'ajouter un en-tête correspondant, décalant alors tout le contenu du fichier situé après. Des emplacements de la zone de code en mémoire ne sont pas utilisés et peuvent servir à injecter de tout petits bout de code, par exemple, entre la dernière instruction d'une fonction et le début de la suivante, il est commun de trouver quelques octets ne servant qu'à aligner le code de la fonction suivante. Ces octets peuvent, par exemple, être remplacés par un branchement.

Dans le but d'utiliser ces tables de délégation dans le processus cible, il est nécessaire de modifier son code binaire. La modification de code binaire est particulièrement difficile et nécessite une grande précision, plus encore si le code est déjà en cours d'exécution. De manière générale, il n'est pas vraiment possible de décaler du code en mémoire sans prendre le risque de corrompre les instructions de branchement direct situées aux alentours. En effet, les branchements directs et les accès mémoire en x86-64 utilisent des décalages par rapport au pointeur d'instruction comme opérande plutôt qu'une adresse absolue. En

effet, dans notre situation, si les accès à la mémoire utilisaient un adressage absolu, l'intégralité de l'adresse serait écrite dans l'instruction, il suffirait alors de la remplacer par la nouvelle. Néanmoins, puisque le jeu d'instructions x86-64 utilise un adressage relatif, une instruction peut référencer de petits décalages avec des instructions plus courtes que pour écrire une adresse complète. Déplacer une instruction ou tout un bloc de base peut amener ces instructions à référencer le mauvais emplacement mémoire (Duck, Gao et Roychoudhury 2020).

De plus, il n'est pas possible d'utiliser uniquement des trampolines, afin de rediriger l'exécution vers des caves de code contenant le code représentant un tobbogan. Certains branchements indirects sont encodés avec bien moins d'octets qu'il est nécessaire pour encoder un saut vers un trampoline. Par exemple, un `call` vers la valeur d'un registre est encodé avec deux octets alors qu'il faut au minimum cinq octets pour encoder un saut direct vers une quelconque adresse. Par conséquent, non seulement tous les branchements indirects du code du programme doivent être modifiés pour intégrer notre solution, mais aussi tout le reste du code doit être adapté afin de ne pas altérer la sémantique du programme. Puisque l'agencement du code doit être modifié, les instructions faisant des accès directs à la mémoire doivent être adaptées également afin de s'assurer qu'elles ciblent toujours la même zone de donnée ou de code qu'auparavant. Pour cette raison, il est particulièrement difficile de modifier les instructions de branchement dans le code sans décaler toutes les instructions suivantes plus loin dans l'espace d'adressage, nécessitant d'allouer plus de mémoire pour stocker le code du programme.

Les trampolines

Les trampolines sont un mécanisme qui permet de rediriger un branchement vers un autre emplacement en changeant le code se trouvant à la destination d'origine de manière à en faire un saut direct vers la nouvelle destination.

2.2.2 Désassemblage et rebasement en mémoire

Puisqu'il est nécessaire de réécrire toute la section `.text` du programme, nous avons décidé d'allouer une quantité de mémoire contigüe suffisante dans le processus cible afin qu'elle puisse contenir l'intégralité des blocs de base rebasés individuellement ainsi que le code modifié par Damas. Cet espace mémoire est appelé `.secure_text`, en référence

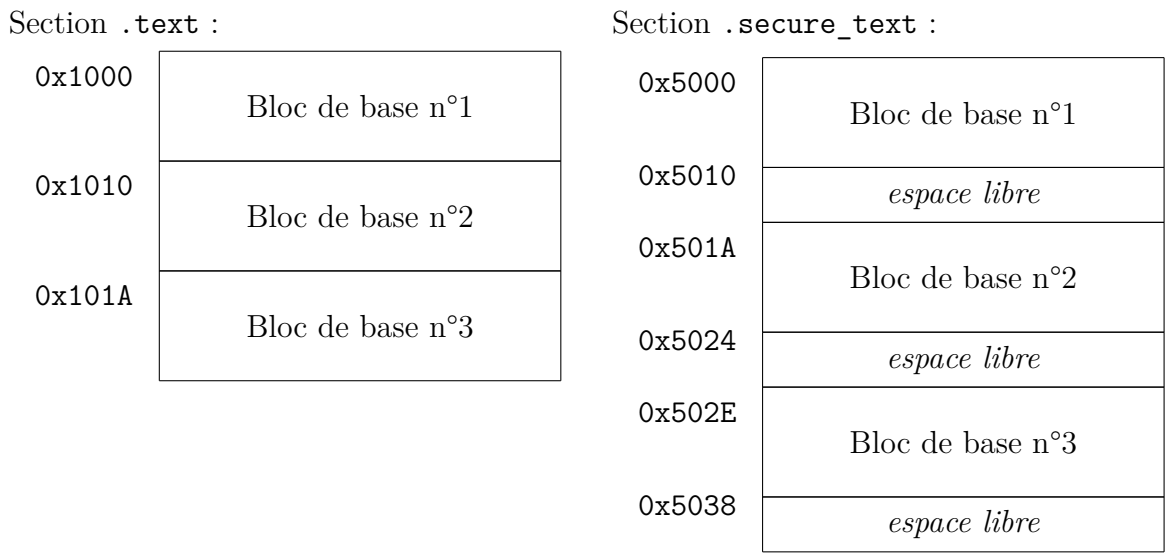

FIGURE 2.1 – Procédure de rebasement du code d'une section `.text` chargée à l'adresse `0x1000` vers une section `.secure_text` chargée à l'adresse `0x5000`

à la section `.text` du fichier ELF. Rebaser chaque bloc de base individuellement permet à notre outil de réserver suffisamment de place entre chaque bloc pour permettre de traduire la dernière instruction des blocs sans empiéter sur les premières instructions du bloc suivant. La figure 2.1 donne un aperçu de cette transformation.

Par ailleurs, déplacer l'intégralité du code en mémoire permet de garder une copie vierge de celui-ci à son emplacement d'origine, autorisant l'utilisateur à retirer les changements opérés par notre outil et rediriger l'exécution du processus sur son code d'origine. Bien que cette hypothèse soit intéressante et potentiellement simple à mettre en place, elle n'a cependant pas été explorée par nos travaux.

Comme expliqué dans la section précédente, déplacer du code dans la mémoire d'un processus tend à invalider les instructions accédant à la mémoire directement, via un décalage par rapport au pointeur d'instruction. Néanmoins, afin que le processus continue de s'exécuter correctement, il est nécessaire de corriger ces instructions afin qu'elles ciblent les mêmes emplacements mémoire qu'auparavant.

2.2.3 Traductions des instructions

Les instructions susceptibles de référencer des emplacements mémoire avec un décalage relatif au pointeur d'instruction peuvent être rangées en deux catégories : les instructions

de branchement direct et les autres. Les deux catégories sont fondamentalement traduites de la même manière : le décalage est ajusté pour que l'instruction pointe au bon emplacement mémoire. Néanmoins, la taille de l'instruction en question peut varier en fonction de la valeur de ce décalage, ce qui est particulièrement le cas des instructions de branchement conditionnel. Cela justifie de distinguer ces deux catégories d'instructions.

Instructions de branchement direct

Les instructions de branchement direct correspondent à toutes les instructions de branchement dont l'opérande est un immédiat. En x86-64, ces immédiats représentent un décalage par rapport au pointeur d'instruction. Par ailleurs, il est important de noter que, lors de l'exécution du branchement, le pointeur d'instruction référence l'instruction juste après le saut, et non l'instruction de saut elle-même. Ainsi, la formule pour calculer l'adresse cible d'un branchement direct est la suivante :

$$adresse_{cible} = adresse_{branchement} + taille_{branchement} + décalage$$

Les instructions de branchement direct peuvent être soit des appels de fonction, soit des sauts. Par définition, il n'est pas possible qu'une instruction de retour de fonction soit un branchement direct, puisqu'il n'est pas possible de savoir statiquement quelle est l'adresse de retour de la fonction. En x86-64, cette adresse est placée sur la pile et elle est dépilée par l'instruction de retour au moment de son exécution.

La traduction des branchements directs est délicate. L'idée directrice est de modifier l'opérande de l'instruction afin que l'adresse cible de celle-ci soit correcte. Par correcte, nous entendons que l'adresse cible de l'instruction de branchement doit être l'adresse de l'instruction cible telle que définie par la sémantique du programme dans le code rebasé en mémoire. Modifier l'opérande de l'instruction de manière à ce que l'adresse cible soit la même que celle d'origine ne suffit pas, il faut dévier le flot de contrôle du programme afin qu'il n'exécute que le code modifié par Damas et non plus son code d'origine. Ainsi, pour traduire une instruction de branchement direct, son adresse cible est calculée, puis l'adresse correspondante dans le code rebasé est récupérée. L'opérande de l'instruction de branchement et le décalage entre la nouvelle instruction traduite et sa nouvelle adresse cible. Ce décalage est obtenue via une soustraction entre l'adresse de la nouvelle instruction de branchement et l'adresse de sa cible.

Néanmoins, la taille de ces instructions est susceptible de varier. En effet, certaines

00000000:	7C	05		jl 0x05
00000000:	0F	8C	EF BE AD DE	jl 0xDEADBEEF
00000003:	7D	FB		jge 0
00123456:	0F	8D	A4 CB ED FF	jge 0

FIGURE 2.2 – Exemples de sauts conditionnels en x86_64

instructions, comme les sauts, existent avec différentes tailles d'opérandes, rendant la traduction plus compliquée. Par exemple, les sauts conditionnels peuvent prendre un immédiat de 8 bits ou de 32 bits comme illustré par la figure 2.2. Par conséquent, si un saut conditionnel sur 8 bits est traduit et que le nouveau décalage calculé dépasse le décalage maximum qui peut être encodé sur 8 bits, il faut en plus traduire cette instruction en un saut sur 32 bits. Cette traduction augmentant la taille de l'instruction de saut, elle nécessite aussi de changer l'opérande de l'instruction qui, comme expliqué plus tôt, dépend aussi de la taille de l'instruction. Heureusement toutefois, les branchements se situent par définition à la fin d'un bloc de base, par conséquent les variations de taille de ces instructions ne peuvent pas impacter les autres instructions du bloc de base. Ainsi, si suffisamment d'espace est réservé entre deux blocs de base, il est possible de modifier les instructions de branchement sans impacter le bloc de base suivant non plus.

Les autres instructions

Certaines instructions référencent des adresses mémoires via un décalage par rapport au pointeur d'instruction mais ne sont pas des branchements. Ces instructions sont très variées car toutes les instructions faisant référence à des adresses utilisent le mode d'adressage relatif (au pointeur d'instruction), par défaut. Ce mode d'adressage est du type *base + décalage*, qui impose que le décalage soit encodé sur 32 bits, comme le montre la figure 2.3.

De ce fait, la taille de ces instructions n'est pas impactée par la valeur du décalage par rapport au pointeur d'instruction. Il n'est pas nécessaire de prendre les mêmes précautions qu'avec les branchements directs. C'est la raison pour laquelle les blocs de bases du programmes sont rebasés en entier avant traduction de leur terminateur (la dernière instruction, généralement un branchement). Le décalage est simplement recalculé et réécrit par-dessus l'ancien décalage dans l'instruction.

00000000:	48 8D	05	05 00 00 00	lea rax, [rip+0x5]
00000007:	48 8D	05	56 34 21 00	lea rax, [rip+0x123456]
0000000E:	48 2B	1D	56 34 21 00	sub rbx, [rip+0x123456]
00000015:	48 03	05	EF BE AD DE	add rax, [rip+0xDEADBEEF]

Figure 2.3 – Exemples d'instructions utilisant le mode d'adressage relatif au pointeur d'instruction en x86_64

2.3 Suppression des branchements indirects

L'un des principaux problèmes auxquels notre approche fait face sont les instructions de branchement indirects. En effet, contrairement aux branchements directs, il est particulièrement difficile de traduire de manière systématique ces instructions afin qu'elles ciblent les bons emplacement dans la mémoire. Le problème réside dans l'idée même du branchement indirect : l'adresse cible du branchement n'est connue qu'à l'exécution du code. Bien souvent, cette adresse n'est pas unique et peut dépendre du contexte d'exécution. C'est typiquement le cas des retours de fonction. Par exemple, une même fonction peut être appelée à plusieurs endroits différents dans le code. Dans ce cas, l'adresse de retour de la fonction déprendra d'où elle aura été appelée.

Un autre exemple sont les pointeurs de fonction. Par exemple, une fonction de tri peut prendre en paramètre une fonction de comparaison, permettant de trier différemment en fonction du besoin. Le code C du listing 9 illustre ce mécanisme. Il s'agit d'un *bubble sort* naïf sur des entiers prenant une fonction de comparaison en paramètre.

```
1  typedef enum ordering_t ordering_t;
2  enum ordering_t { LT, EQ, GT };
3
4  typedef ordering_t (*cmp_func)(int, int);
5
6  void sort_by(int* array, int length, cmp_func cmp) {
7    int i, j;
8
9    for (i = 0; i < length; ++i) {
10     for (j = 0; j < length; ++j) {
11       if (cmp(array[i], array[i+1]) == GT) {
12         swap(&array[i], &array[i+1]);
13       }
14     }
15   }
16 }
```

Listing 9: Fonction de tri paramétrée par une fonction de comparaison

Lors de l'appel de cette fonction `sort_by`, l'adresse de la fonction de comparaison est donnée en paramètre. Pendant l'exécution de la fonction de tri, la fonction `cmp` est appelée au moyen de ce pointeur, stocké dans un registre. Ainsi, l'appel à `cmp` sera une instruction `call` indirect.

La difficulté est donc de ne pas savoir à l'avance quelle sera l'adresse cible de l'instruction de branchement indirect. Il y a deux principales approches permettant néanmoins de rediriger un branchement indirect : modifier toutes les références aux adresses cibles dans le code et les données du programme ou transformer l'instruction de branchement en un code plus complexe réalisant la redirection pendant l'exécution.

La première approche suit la même philosophie que l'approche présentée à la sous-section 2.2.3, à savoir modifier le code du programme ou ses données afin que le programme puisse être rebasé quasiment à l'identique en conservant sa sémantique. Dans le cas des branchements indirects, il ne serait bien souvent pas question de modifier des décalages par rapport au pointeur d'instruction, mais plutôt de modifier les données du programme et plus précisément des tables de sauts ou des pointeurs écrits en dur dans le code. Cette approche souffre néanmoins de plusieurs défauts. Tout d'abord, afin de

modifier le bon emplacement mémoire sensé contenir l'adresse cible d'un branchement, il est nécessaire de savoir de quel emplacement mémoire il s'agit. En effet, puisque le branchement est indirect, son opérande n'est pas l'adresse cible mais un emplacement où la machine trouvera l'adresse vers laquelle rediriger le saut. Or, en principe cette valeur n'est connue qu'à exécution, il n'est donc pas toujours possible de savoir ni où ni quoi modifier pour rediriger correctement l'exécution. De plus, l'opérande du branchement est rarement elle-même juste un pointeur écrit en dur. Il s'agit souvent d'une expression utilisant des registres et des pointeurs dont les valeurs ne sont pas toujours prédictibles. En clair, il n'est pas possible de prédire systématiquement de manière fiable les emplacements où se trouvent les adresses cibles des branchements.

La seconde approche est plus réaliste. Elle ne modifie pas les données du programme et laisse le processus calculer l'adresse cible du branchement normalement. Elle utilise ensuite cette information dans le code instrumenté, au niveau de chaque branchement, afin d'effectuer la redirection. Ainsi, au lieu de deviner où le branchement peut mener, une indirection se chargeant de rediriger le flot de contrôle vers la bonne instruction dans `.secure_text` est ajoutée. Nous avons privilégié cette approche dans nos travaux puisqu'elle nous permet de nous affranchir d'analyses coûteuses et approximatives.

En plus de rediriger l'exécution du processus vers un nouveau flot de contrôle modifié par notre outil, nous essayons d'en garantir l'intégrité. Pour ce faire, nous ne nous contentons pas de rediriger les branchements indirects, nous nous assurons que leur cible est bien conforme à la sémantique du programme. Nous avons adapté pour cela la solution proposée par CDI. Dans la sous-section suivante, nous décrivons en détail comment fonctionne cette solution et expliquons pourquoi il est difficile de s'en servir dans un contexte dynamique, tel que celui dans lequel s'inscrivent nos travaux. Puis, dans la sous-section 2.3.3 nous expliquons les changements apportés à l'approche de CDI et présentons les difficultés rencontrées pendant le développement ainsi que les solutions trouvées pour y faire face.

2.3.1 Les tobbogans de Control-Data Isolation

Control-Data Isolation propose une approche de l'intégrité du flot de contrôle intéressante. Contrairement aux approches de type CFI, il n'est pas question de valider l'intégrité des branchements indirects. Cette approche supprime l'origine du problème, à savoir l'existence même des branchements indirects. En clair, les branchements indirects sont transformés en branchements directs.

Le programme d'exemple du listing 10 définit deux fonctions `f` et `g` qui affichent un

message à l'écran. Dans la fonction `main`, une de ces deux fonctions est appelée indirectement grâce au pointeur de fonction `ptr`. Ce pointeur est initialisé par la fonction `chooseBetweenFandG` et l'on suppose que cette initialisation est correcte (c'est-à-dire, que `ptr` pointe vers `f` ou `g` à l'issu de l'initialisation).

```
1  void f() {
2    puts("Hello");
3  }
4
5  void g() {
6    puts("Good morning");
7  }
8
9  int main(void) {
10   void (*ptr)() = chooseBetweenFandG();
11   ptr();
12
13   return 0;
14 }
```

Listing 10: Exemple d'utilisation d'un pointeur de fonction

Une fois compilé, l'initialisation du pointeur, ainsi que l'appel indirect à la fonction correspondante dans la fonction `main`, ressemble au code du listing 11 (on suppose que la compilation est réalisée sans optimisations).

```
1    mov    eax,0x0
2    call   0x115f ; <chooseBetweenFandG>
3    mov    QWORD PTR [rbp-0x8],rax
4    mov    rdx,QWORD PTR [rbp-0x8]
5    mov    eax,0x0
6    call   rdx
```

Listing 11: Appel indirect à la fonction référencée par`ptr`

Le pointeur de fonction est stocké dans la pile à l'emplacement `[rbp-0x8]`. Sa valeur est copiée dans le registre `rdx` et la fonction est appelée via l'instruction d'appel indirect

`call rdx`. Afin de supprimer cet appel indirect, CDI utilise le CFG construit par le compilateur et remplace l'instruction d'appel de fonction par une structure de contrôle appelé un tobbogan.

Un tobbogan est un enchaînement de comparaisons et de branchements qui permet de s'assurer que la valeur de l'opérande du branchement indirect correspond à une adresse cible valide et de brancher directement au bon endroit. Dans l'exemple du listing 12, le tobbogan pourrait ressembler au code suivant. Dans cet extrait de code, on considère que la fonction `under_attack` est une fonction introduite par CDI afin de gérer le cas où notre pointeur n'a pas une valeur valide et où on peut légitimement considérer que le processus est victime d'une attaque par détournement du flot de contrôle :

```
1 if (ptr == f) {
2   f();
3 } else if (ptr == g) {
4   g();
5 } else {
6   under_attack();
7 }
```

Listing 12: Illustration d'un tobbogan simple

De cette manière, le flot de contrôle et le flot de données sont découplés. Bien que les données du processus (ici le pointeur `ptr`) permettent de décider de la suite de l'exécution, à aucun moment la valeur même du pointeur n'est utilisée comme cible d'un branchement. Tous les branchements deviennent directs, garantissant une plus grande fiabilité du flot de contrôle.

Afin de mettre en place ce système, CDI utilise le CFG généré par le compilateur. Ce CFG est complet et minimal, ce qui signifie que toutes les transitions du graphe sont des branchements valides du programme et que tous les branchements du programme apparaissent comme transition dans le graphe. Grâce à ce CFG, CDI est capable de définir précisément les cibles possibles de chaque branchement dans le programme et de générer un tobbogan pour chaque branchement indirect du programme.

Dans notre cas, en l'absence d'informations fournies par le compilateur, il est particulièrement difficile de résoudre les branchements indirects et donc d'obtenir un CFG complet. Typiquement, les appels de fonctions indirects et les retours de fonctions ne

peuvent pas toujours être précisément résolus. Par conséquent, la liste des adresses cibles potentielles peut devenir trop longue pour écrire un tobbogan pour chaque instruction de branchement indirect. En effet, si un toboggan déraisonnablement grand était copié à la place de chaque instruction de branchement indirect, la consommation mémoire du processus pourrait être un problème majeur. De plus, les adresses cibles potentielles sans l'aide d'un CFG complet seront en grande majorité les mêmes pour différents branchements indirects. Par exemple, sans graphe d'appels des fonctions, il n'est pas possible de savoir quelles fonctions appellent quelles autres fonctions et donc il n'est pas possible de connaitre les adresses de retour d'une instruction `ret`. Ainsi, toutes les instructions `ret` pourraient considérer les mêmes adresses comme valides, à savoir toute instruction située juste après une instruction `call`.

Pour ces raisons, notre approche ne génère pas de tobbogan pour chaque branchement indirect. Afin de réduire au maximum l'impact de notre approche sur la taille du code et de garantir que le programme cible conserve sa sémantique initiale, nous construisons des classes d'équivalence entre les différentes instructions de branchement indirect afin de factoriser le code généré pour la gestion de ces branchements.

2.3.2 Les classes d'équivalence de branchements

L'une des particularités de notre approche tient en la définition des classes d'équivalences de branchements indirects. À défaut de pouvoir identifier précisément les cibles potentielles de chaque branchement indirect, notre approche essaie de regrouper ces branchements dans des classes d'équivalence afin de leur assigner une liste commune d'adresses cibles potentielles.

Une classe d'équivalence de branchements respecte deux règles :

— Toutes les adresses qu'un branchement appartenant à la classe d'équivalence peut atteindre sont contenues dans la liste des adresses cibles potentielles définies par la classe d'équivalence.

— La liste des adresses cibles potentielles d'une classe d'équivalence est la plus réduite possible.

La raison pour laquelle il est souhaitable qu'une classe d'équivalence soit la plus réduite possible est que toute transition superflue dans le CFG qu'elles permettent d'atteindre est un risque en terme de sécurité. En effet, chaque adresse cible potentielle considérée valide par notre approche, mais qui n'est pas une adresse cible valide selon le CFG construit par le compilateur, est un potentiel faux négatif. Ainsi, réduire la taille des classes d'équiva-

lences permet de réduire la surface d'attaque qui permettrait à un attaquant d'abuser des imprécisions de notre approche.

La difficulté réside donc dans la définition des classes d'équivalences et dans l'identification précise des adresses cibles potentielles de chaque instruction de branchement indirect.

Durant nos travaux nous avons construit des classes d'équivalences en nous basant sur les hypothèses suivantes :

— H1 : Les appels de fonction (les instructions `call`) ciblent la première instruction d'une fonction.
— H2 : Les retours de fonction (l'instruction `ret`) ciblent l'instruction juste après un appel de fonction.
— H3 : Les sauts (les instructions `jmp`) peuvent cibler n'importe quelle instruction au sein de la même fonction ou la première instruction d'une fonction.

Les hypothèses H1 et H2 sont assez intuitives. Puisqu'on suppose que le code du processus cible a été généré par un compilateur et qu'il n'a pas été obfusqué, les fonctions ont un seul point d'entrée. Lorsqu'une fonction est appelée, l'instruction d'appel de fonction cible nécessairement cette adresse. Ainsi, il est tout à fait raisonnable de penser que si un `call` cible une adresse qui n'est pas la première instruction d'une fonction, le flot de contrôle a alors été dévié. De la même manière, si retour de fonction il y a, nécessairement, la fonction a été appelée. Puisque le code n'est pas obfusqué, la fonction a été appelée via une instruction `call` [2].

L'hypothèse H3 est plus pragmatique. Puisque nous supposons le code binaire issu d'un compilateur classique sans obfuscation, il est raisonnable de penser que le flot de contrôle du code binaire est structuré d'une manière similaire à son code source. Ainsi, il est raisonnable de penser que des sauts inconditionnels servent essentiellement au flot de contrôle interne aux fonctions et non à sauter d'une fonction à une autre (à l'exception des optimisations des appels de sauts récursifs). Bien que nous n'ayons jamais prouvé formellement cette hypothèse, nos différentes expérimentations semblent toutefois la confirmer. H3 admet aussi qu'un `jmp` puisse cibler le début d'une fonction. Il s'agit du cas des optimisations des appels récursifs terminaux.

2. L'optimisation des appels récursifs terminaux (*tail-call recursion optimization*) ne fait pas exception à cette règle. En effet, l'intérêt de remplacer des instructions `call` par des `jmp` est justement de faire l'économie d'autant de retours de fonction (en plus d'optimiser l'utilisation de la pile) et par conséquent, l'unique `ret` de la chaîne d'appels de fonctions répond au tout premier `call` de cette même chaîne.

Optimisation des appels récursifs terminaux

L'optimisation des appels récursifs terminaux est une optimisation servant essentiellement à améliorer l'utilisation de la pile par les fonctions récursives. L'idée est simple, si la fonction se termine par un appel de fonction (elle-même ou une autre) et retourne aussitôt après, il est possible de remplacer le `call` par un `jmp`. En effet, puisque la fonction retourne aussitôt après son appel, la *stack frame* qui lui est dédiée n'est plus utile, il est alors possible de l'écraser. Cette optimisation sert à éviter des débordements de pile (*stack overflow*) liés à une trop grande chaîne d'appels récursifs mais elle a aussi l'avantage de supprimer tous les retours de fonction intermédiaires, ne nécessitant ainsi qu'une seule instruction `ret` pour retourner de toute la chaîne d'appels en un seul coup.

Les fonctions `setjmp` et `longjmp` ne sont pas prises en compte dans notre approche. Ces fonctions permettent des sauts non-locaux dans du code écrit en C. Ces fonctions servent explicitement à sauter de l'intérieur d'une fonction à une instruction d'une autre fonction, sans utiliser la sémantique d'appel ou de retour de fonction. Ces fonctions invalident par définition notre hypothèse H3 et nécessiteraient un travail de recherche supplémentaire pour en assurer correctement l'intégrité. Néanmoins, l'existence de ces fonctions dans la bibliothèque standard du C nous conforte dans la validité de H3 [3].

Ces hypothèses nous permettent de définir trois types de classes d'équivalence. La classe des appels de fonction, la classe des retours de fonctions et enfin toutes les classes d'équivalence des sauts indirects, une par fonction. En l'absence de plus d'informations sur le flot de contrôle du programme, il est difficile de discriminer davantage les cibles potentielles des instructions `call` et `ret` au delà des instructions définies par les hypothèses H1 et H2. C'est la raison pour laquelle, dans le cadre de nos travaux, nous avons défini une classe d'équivalence pour toutes les instructions d'appel de fonction et une autre pour toutes les instructions de retour de fonction. Notre hypothèse H3 nous permet de raffiner un peu plus les cibles potentielles des sauts indirects, évitant de rendre valide toutes les instructions du programme. En effet, il est possible de définir une classe d'équivalence

3. S'il est nécessaire de recourir à ces fonctions, alors les programmes écrits dans des langages tels que C ou de plus haut niveau ne permettent pas nativement de faire des sauts non-locaux.

pour tous les sauts situés au sein d'une même fonction.

2.3.3 Les tables de délégation

Comme expliqué précédemment, il n'est pas raisonnable de remplacer chaque branchement indirect du programme par un tobbogan en utilisant notre approche, sans faire exploser la taille du code. C'est la raison pour laquelle nous avons défini des classes d'équivalences entre les branchements, comme expliqué à la sous-section 2.3.2. L'intérêt de ces classes d'équivalence est de rediriger le flot de contrôle de chaque branchement dans la même classe d'équivalence vers un seul et même tobbogan.

Néanmoins, cette approche est plus compliquée à mettre en place qu'il n'y parait. En effet, contrairement à un tobbogan classique, l'emplacement mémoire contenant l'adresse à comparer n'est pas toujours le même. Un tobbogan classique remplace un seul branchement indirect, son opérande est donc connue. Il est alors possible de l'utiliser directement dans le code du tobbogan. Par exemple, l'instruction `call rax` pourrait être remplacée par le code du listing 13.

```
1  _case_1:
2    cmp rax, func1
3    jne _case_2
4    call func1
5  _case_2:
6    cmp rax, func2
7    jne _case_3:
8    call func2
9  _case_3:
10   ;; ...
```

Listing 13: Remplacement d'une instruction `call rax` par un toboggan

Une instruction comme `call rdx` pourrait être remplacée par le même tobbogan mais en utilisant `rdx` comme opérande pour les comparaisons plutôt que `rax`.

Toutefois, ceci n'est pas possible avec notre approche. En effet, deux instructions utilisant deux opérandes différentes pourraient faire partie de la même classe d'équivalence. Pour cette raison, nous avons modifié la structure des tobbogans afin d'y ajouter un prologue dont l'objectif est de définir un registre de travail (*preferred register*) pour le tobbo-

gan. Ce prologue prend la forme d'un répartiteur dont chaque entrée correspond à une des opérandes possibles apparaissant dans les branchements appartenant à la classe d'équivalence associée au tobbogan. Chaque entrée du répartiteur copie la valeur de l'opérande du branchement dans le registre de travail puis saute à l'adresse de début du tobbogan afin d'exécuter la séquence de comparaisons/branchements attendue.

Par exemple, soit une classe d'équivalence comportant les instructions du listing 14.

```
1  call rdx
2  call QWORD PTR [rcx+0x10]
3  call QWORD PTR [rdi+rcx*2]
```

Listing 14: Classe d'équivalence de branchements

Le prologue du tobbogan permettant de remplacer ces instructions pourra être celui présenté au listing 15, en considérant que `rax` sera utilisé comme registre de travail [4].

```
1  _entry1:
2    mov rax,rdx
3    jmp _sled
4  _entry2:
5    mov rax,QWORD PTR [rcx+0x10]
6    jmp _sled
7  _entry3:
8    mov rax,QWORD PTR [rdi+rcx*2]
9    jmp _sled
10 _sled:
```

Listing 15: Prologue de la table de délégation de la classe d'équivalence du listing 14

Grâce à ces prologues, toutes les instructions de la classe d'équivalence pourront utiliser le même tobbogan. Il suffit alors de rediriger le flot de contrôle vers la bonne entrée du prologue et non plus vers le début du tobbogan. Les instructions données en exemples précédemment seraient traduites comme montré au listing 16.

4. Ce choix est pour l'instant arbitraire et ne sert qu'à illustrer nos propos. Les registres de travail utilisés dans DAMAS sont décrits plus loin dans ce chapitre.

```
1 call _entry1
2 call _entry2
3 call _entry3
```

Listing 16: Instructions d'appel de fonction redirigées vers les points d'entrée de la table de délégation

Il existe toutefois un cas particulier, celui où l'opérande de l'instruction de branchement est égal au registre de travail du tobbogan. Dans notre exemple, il s'agirait de l'instruction `call rax`. Afin d'éviter de rajouter une instruction `mov` et un saut qui seraient inutiles, aucune entrée n'est ajoutée dans le prologue pour cette instruction, le point d'entrée du tobbogan (ici le symbole `_sled`) est utilisé directement.

Nous avons nommé la structure décrite jusqu'à présent *table de délégation* afin de faire la différence avec les tobbogans de CDI. Comme nous le verrons par la suite, les différences entre les tobbogans et les tables de délégations ne se limitent ni à la présence d'un prologue, ni à la façon dont on y accède. En effet, les trois instructions de branchement pouvant être indirects, `call`, `ret` et `jmp` ne sont pas gérés de la même manière. Par exemple, pour gérer un retour de fonction, il est nécessaire d'ajuster manuellement le registre de pile (`rsp`), ce qui n'est pas utile pour un appel de fonction [5].

La table d'appels des fonctions

Dans le but de remplacer les instructions `call` du processus cible, une seule et unique table de délégation est utilisée. Tel qu'expliqué précédemment, en l'absence d'analyses statiques très poussées permettant de définir un graphe d'appel complet (ou du moins un sur-ensemble restreint de celui-ci), tous les appels de fonctions du programme sont regroupés dans la même classe d'équivalence. Cette table se distingue essentiellement par la taille de son prologue et son nombre d'entrées. En effet, puisque cette table regroupe tous les `call` du programme, le nombre d'entrées de son prologue augmente drastiquement avec la taille du code.

Le calcul de la taille du code binaire d'une table de délégation est important puisqu'il sert à connaître la taille du tampon à allouer dans le processus cible d'une manière suffisamment précise, pour ne pas gaspiller de mémoire en allouant des tampons exagérément

5. En effet, il est possible de traduire un `call` indirect en `call` direct vers une table de délégation, contrairement à l'instruction `ret` qui est indirect par essence, nécessitant une traduction en une autre séquence d'instructions.

longs. Or la taille du prologue est la partie la plus compliquée à calculer dans la table de délégation. Il y a plusieurs raisons à cela.

D'abord, les instructions `mov` peuvent prendre des opérandes très diverses, ce qui rend la taille de ces instructions très variable. De plus, x86-64 est un jeu d'instruction à taille variable complexe dont la taille des instructions n'est pas toujours intuitive [6]. Ensuite, puisque chaque entrée du prologue contient un saut en direction de la fin du même prologue, la taille de ces entrées peut varier en fonction du décalage entre l'adresse du saut et l'adresse de sa destination. En effet, en x86-64, il est possible d'encoder un saut inconditionnel direct sur 8 bits ou sur 32 bits.

Contrairement aux autres tables décrites par la suite, un calcul précis de la taille du prologue de la table des appels de fonctions est important. Les autres tables ont généralement peu d'entrées, ce qui permettrait de calculer une approximation conservatrice de la taille du prologue. Pour la table des appels, étant donné son nombre important d'entrées, une approximation par le haut de la taille du prologue pourrait amener à un gaspillage de mémoire non négligeable [7].

Concernant le registre de travail, il existe un bon candidat pour la table des appels de fonctions. En effet, le registre `rax` est utilisé dans de nombreuses opérations en x86-64, ne correspond à aucun paramètre de fonction et a le rôle de valeur de retour des fonctions. Il peut être utilisé comme registre de travail sans nécessiter de le sauvegarder avant d'entrer dans la table et de le restaurer avant d'en sortir.

Le tobbogan de la table des appels de fonctions, quant à lui, est assez classique, il se compose de comparaisons entre le registre de travail de la table et les adresses cibles potentielles, ainsi que de sauts vers ces adresses. La différence entre un tobbogan de CDI et un tobbogan de nos tables est que le saut ne dirige pas le flot de contrôle vers l'adresse identifiée par la comparaison précédente, mais vers l'adresse rebasée correspondante. En effet, comme expliqué précédemment, l'adresse calculée par le processus, et qui est stockée dans le registre de travail, est l'adresse cible de l'appel indirect telle qu'elle aurait été calculée sans l'intervention de notre outil. Il est donc logique que les comparaisons dans le tobbogan se fassent avec ces adresses. Néanmoins, puisqu'il est question de rediriger le flot de contrôle en direction de notre version sécurisée du code, les sauts ne ciblent pas les

6. Parfois, une différence de registre peut faire varier la taille de l'instruction. Par exemple, pour une instruction de la forme `mov rax, offset(reg)`, si `reg` est `r12` ou `rsp`, l'encodage de l'instruction prendra un octet de plus.

7. À noter que nous avons été très prudent dans notre utilisation de la mémoire du processus cible. Il est possible aussi d'allouer une très grande quantité de mémoire virtuelle via `mmap` et de n'en utiliser que le nécessaire. Les pages de mémoires qui se sont pas utilisées ne seront pas allouées physiquement.

adresses originales mais les adresses rebasées. Le code suivant est un exemple de tobbogan d'une table de délégation pour les appels de fonctions :

```
1 cmp eax, _func1
2 je _relocated_func1
3 cmp eax, _func2
4 je _relocated_func2
5 cmp eax, _func3
6 je _relocated_func3
7 jmp _error_handler
```

Ce tobbogan reconnaît les adresses de début de toutes les fonctions du binaire principal du programme instrumenté. Il reconnait aussi les entrées de la PLT. La PLT est une table permettant de résoudre les appels de fonctions vers des bibliothèques pour les exécutables dont le code est indépendant de sa position. En effet, le programme peut effectuer un appel indirect à une fonction dans une bibliothèque, c'est pourquoi il est nécessaire de s'intéresser aussi à la PLT. Les entrées de la PLT sont un cas compliqué. En effet, il serait intéressant de remplacer les références à la PLT dans les tables de délégations par les adresses cibles déjà résolues des bibliothèques. Toutefois, ce n'est pas aussi simple. En effet, par défaut, l'éditeur de liens dynamique (*dynamic linker*) de Linux travaille paresseusement et ne résout les adresses des symboles qu'à leur première utilisation. Ainsi, au moment de l'exécution où notre solution est attachée au processus cible, si un symbole n'est pas encore résolu, il n'est pas possible de prédire son adresse future dans l'espace d'adressage.

Par ailleurs, puisque les entrées de la PLT sont modifiées dynamiquement par l'éditeur de lien dynamique afin de relier (paresseusement) ces entrées au symbole correspondant, il est particulièrement difficile de rebaser la PLT afin de supprimer des sauts indirects que son utilisation implique. Pour cette raison, nous avons décidé de ne pas la modifier et de laisser la table de délégation rediriger le flot de contrôle vers la PLT sans autre indirection. Ce qui implique que notre implémentation actuelle ne protège que le binaire principal de l'application mais pas les bibliothèques qu'elle utilise.

Les tables de retours de fonctions

La table des retours de fonctions est un peu plus complexe que la table des appels de fonctions. Comme énoncé précédemment, la traduction des instructions `ret` en saut

direct vers une table de délégation impose une manipulation manuelle de la pile. En effet, il est nécessaire d'ajuster manuellement le registre de pile, `rsp`. De plus, il est nécessaire aussi de s'intéresser au registre de travail. Dans le cas de la table des appels de fonction, il n'était pas nécessaire de sauvegarder ce registre. Puisque `rax` n'est pas utilisé comme paramètre de fonction et n'est pas considéré comme sauvegardé par la fonction appelante (*caller-saved*), il est possible de l'écraser sans conséquence. Dans le cas de la table de retours de fonctions, ce registre sert de valeur de retour. De plus, aucun autre registre n'est vraiment disponible puisque les autres registres peuvent être utilisés par la suite.

Par exemple, si deux fonctions sont appelées l'une après l'autre avec les mêmes paramètres et que la première fonction ne modifie pas les registres contenant les valeurs de ces paramètres [8], alors le code assembleur correspondant pourra être deux instructions `call` à la suite sans réinitialiser les registres. Ce choix d'implémentation permet un gain de temps d'exécution et d'espace (en taille de code).

Ainsi, il est difficile d'émettre des hypothèses concernant les registres pouvant servir de registre de travail sans nécessiter une sauvegarde et une restauration. C'est pourquoi la table de délégation des retours sauvegarde systématiquement le registre de travail. Le registre utilisé à cet effet par notre implémentation est `r11`. Il s'agit d'un choix arbitraire parmi tous les registres disponibles. Au sein de notre implémentation, le registre `r11` est sauvegardé derrière le pointeur de pile, à l'adresse pointée par `rsp - 8`. En théorie, cet emplacement n'est pas censé être utilisé et représente un emplacement de choix pour y sauvegarder temporairement des données écrasées par nos manipulations.

La figure 2.4 donne un aperçu de la configuration de la pile lorsque le flot de contrôle entre dans la table de délégation des retours de fonction. Contrairement à la table des appels de fonction, la table des retours n'a qu'une entrée dans son prologue. En effet, l'instruction `ret` ne prend pas l'adresse cible comme opérande. Le plus souvent, cette instruction est utilisée sans opérande, autrement, cette opérande désigne une valeur à soustraire du pointeur de pile avant de retourner de la fonction. Une fois cette soustraction effectuée, si besoin, l'adresse de retour se trouve systématiquement à l'adresse pointée par le pointeur de pile. Ainsi, la seule entrée du prologue consiste à copier cette valeur dans le registre de travail, comme le montre le listing 17.

8. D'un point de vue extérieur. Si la fonction modifie ces registres mais les restaure avant de retourner, ils sont considérés comme non-modifiés.

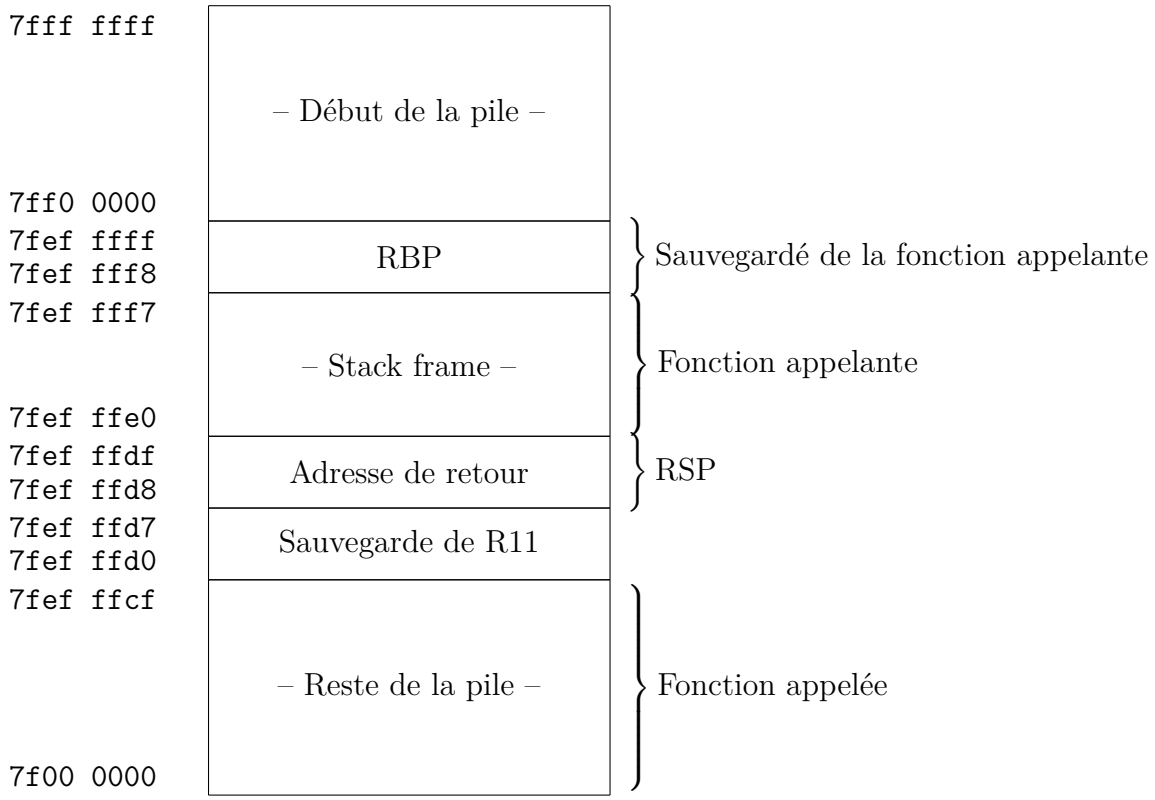

Figure 2.4 – Configuration de la pile lors de l'entrée dans la table des retours de fonction

```
1 _ret_dispatch_table:
2   mov QWORD PTR [rsp-8], r11
3   mov r11, QWORD PTR [rsp]
4   ;; inutile de sauter
5 _sled:
6   ;; ...
```

Listing 17: Prologue de la table de délégation des retours de fonction

Le tobbogan de cette table est plus complexe que celui de la table des appels de fonctions. En effet, bien que l'idée de base reste la même, à savoir une comparaison avec le registre de travail et un saut, il existe plusieurs contraintes à respecter. Tout d'abord, il est nécessaire d'ajuster le pointeur de pile pour respecter la sémantique du retour de fonction. Enfin, il faut s'attarder sur le décalage entre l'adresse du saut et l'adresse de retour de la fonction. En effet, selon où se trouve la fonction appelante dans l'espace d'adressage, il est possible que le décalage entre la table de délégation et celle-ci ne soit pas encodable avec 32 bits. C'est typiquement le cas si la fonction appelante se trouve dans une bibliothèque. Puisqu'il n'existe pas en x86_64 de branchement direct dont l'opérande est un décalage sur 64 bits, il est nécessaire ici de tricher. En effet, l'adresse cible du saut est écrit sur la pile au niveau du pointeur de pile et un saut indirect est effectué. Puisque cette valeur est écrite en dur dans le code injecté par notre solution et qu'elle est utilisée juste après avoir été écrite sur la pile, l'adresse cible du saut indirect ne peut pas être contrôlée par un utilisateur mal intentionné. En substance, un cas du tobbogan de la table des retours de fonctions ressemble au code suivant du listing 18.

Les tables de sauts inconditionnels

Les tables de délégations permettant de rediriger les sauts inconditionnels sont les plus complexes. La complexité de ces tables vient d'un compromis entre trois facteurs : les performances, la précision du désassemblage et la sécurité.

En effet, comme précisé dans la sous-section 2.3.2, selon l'hypothèse H3, toutes les instructions de la fonction contenant l'instruction de saut sont des cibles valides. Le nombre de cibles potentielles peut être très grand, ce qui implique un nombre important de comparaisons et de sauts dans le tobbogan. En fonction du CFG de la fonction, il est tout à fait possible que les cibles réelles de ces sauts ne soient pas les premières à apparaître dans le tobbogan, imposant une perte de performances rédibitoire.

```
1  _this_case:
2    cmp rax, _ret_addr1
3    jnz _next_case
4    ;; Le pointeur de pile est ajusté.
5    add rsp, 8
6    ;; L'adresse de retour est mise dans rsp - 8.
7    movabs r11, _rebased_ret_addr1
8    mov QWORD PTR [rsp-0x8], r11
9    ;; Le registre de travail est restauré.
10   mov r11, QWORD PTR [rsp-0x10]
11   jmp QWORD PTR [rsp-0x8]
12 _next_case:
13   ;; ...
```

Listing 18: Code d'un cas d'une table de retour de fonction

Un exemple parlant est le cas des `switch`. En effet, les `switch` comportant un nombre important de cas sont compilés sous la forme d'un saut indirect vers une entrée d'une table de saut. Le problème est que puisque ce `switch` est grand, pour atteindre ses derniers cas, le tobbogan doit néanmoins être parcouru en entier. Pour peu que le code à l'intérieur des cas du `switch` ne soit pas restreint, la pénalité en performance peut être catastrophique. Si le `switch` se trouve en fin de fonction, le problème est d'autant plus grave puisque le registre de travail sera systématiquement comparé à tous les blocs de base de la fonction avant ceux du `switch` dont on sait qu'il contient l'adresse cible du saut.

De plus, bien que la méthode employée par notre implémentation pour désassembler le code du programme se soit montrée très efficace, nous sommes conscient du fait que le désassemblage de code binaire est un problème indécidable et donc que le résultat obtenu n'est pas forcément parfait (Andriesse et al. 2016). Par conséquent, il est possible que certains blocs de base obtenus soient le résultat d'un désassemblage dont l'adresse de début est erronée (décalée de quelques octets par rapport au bloc de base attendu). Même si l'adresse de début du bloc est décalé, il est très probable que les instructions qui suivent soient correctes puisque le désassembleur aura tendance à se réaligner avec le code attendu. Dans de telles circonstances, s'appuyer sur l'adresse exacte des instructions d'un bloc pour valider la cible d'un saut peut s'avérer périlleux.

Une solution moins risquée, permettant d'éviter d'éventuels faux positifs, consiste en un test de bornes du bloc de base. La simple comparaison entre deux adresses, utilisée dans les tables d'appel et de retour de fonctions, devient ici un test s'assurant que l'adresse

ciblée est bien contenue dans un bloc de code. En première intention, un cas d'un tobbogan dans une table de délégation des sauts inconditionnels pourrait ressembler au code du listing 19.

```
1  _this_case:
2    ;; rax < _bb1_start ?
3    cmp rax, _bb1_start
4    jl _next_case
5    ;; _bb1_end <= rax ?
6    cmp rax, _bb1_end
7    jge _next_case
8    ;; _bb1_start <= rax < _bb1_end !
9    jmp rax
10 _next_case:
11   ;; ...
```

Listing 19: Code d'un cas d'une table de délégation d'un saut inconditionnel

Toutefois, les tables pour les sauts font face à d'autres contraintes. Premièrement, tout comme les autres tables, elles ne servent pas uniquement à tester la validité d'un branchement, elles permettent aussi de rediriger le flot de contrôle en direction du code rebasé, qui est débarrassé de ses branchements indirects. Dans le cas précis des tables des sauts, un test de bornes est effectué à la place d'une comparaison avec une adresse précise. Le problème que pose ce test est que si la valeur contenue dans le registre de travail se trouve dans les bornes d'un bloc de base, il n'existe pas d'adresse fixe à laquelle rediriger le saut, il en existe autant que d'adresses entre ces mêmes bornes. Ainsi, au lieu de rediriger le saut vers une adresse fixe (telle que l'adresse de début du bloc), l'adresse rebasée correspondant à l'adresse cible originelle du saut doit être recalculée par la table.

Soit $bb_{originel}$ l'adresse de début du bloc de base considéré dans le code originel et bb_{rebase} l'adresse de début du bloc de base rebasé correspondant. Pour une adresse *cible* dans le code originel, l'adresse correspondante $cible_{rebase}$ est calculée grâce à la formule suivante :

$$cible_{rebase} = cible - bb_{originel} + bb_{rebase}$$

Cette formule permet de calculer le décalage par rapport au début du bloc de base ori-

ginel de l'adresse cible et de l'ajouter à l'adresse de début du bloc de base rebasé. Comme expliqué dans la section 2.2, la taille des instructions au sein d'un bloc de base ne change pas pendant le processus de rebasement à l'exception de la toute dernière instruction. Par conséquent, le décalage entre l'adresse de début du bloc de base et l'adresse d'une instruction dans ce bloc ne change pas avec le rebasement. Ainsi, l'instruction cible du saut peut être retrouvée en calculant ce décalage. De plus, les constantes $bb_{originel}$ et bb_{rebase} sont connues à la fin de la procédure de rebasement, ne laissant plus que *cible* comme variable connue uniquement à l'exécution : il s'agit de la valeur contenue dans le registre de travail. Afin d'économiser une instruction, on pose la constante $decalage = bb_{rebase} - bb_{originel}$, permettant de définir $cible_{rebase}$ avec cette formule :

$$cible_{rebase} = cible + decalage$$

Le code du tobbogan devient donc celui du listing 20.

```
1 _this_case:
2   cmp eax, _bb1_start
3   jl _next_case
4   cmp eax, _bb1_end
5   jge _next_case
6   add rax, decalage
7   jmp rax
8 _next_case:
9   ;; ...
```

Listing 20: Code du tobbogan d'une table de délégation de saut inconditionnel

De plus, tout comme pour la table des retours de fonctions, aucun registre ne fait un candidat idéal pour servir de registre de travail. En effet, dans le cas des sauts inconditionnels, rien ne permet de déterminer facilement l'utilisation faite des différents registres. Ainsi, il est nécessaire encore une fois de sauvegarder le registre de travail en entrant dans la table et le restaurer avant d'en sortir. Notre implémentation utilise le registre `rax` comme registre de travail et sauvegarde sa valeur à l'adresse pointée par `rsp - 16`, afin de laisser l'adresse pointée par `rsp - 8` disponible pour une raison expliquée juste après. Puisque le registre de travail est restauré avant le saut, sa valeur est copiée dans `rsp - 24` et le saut utilise cette adresse comme opérande. La configuration de la pile juste avant de

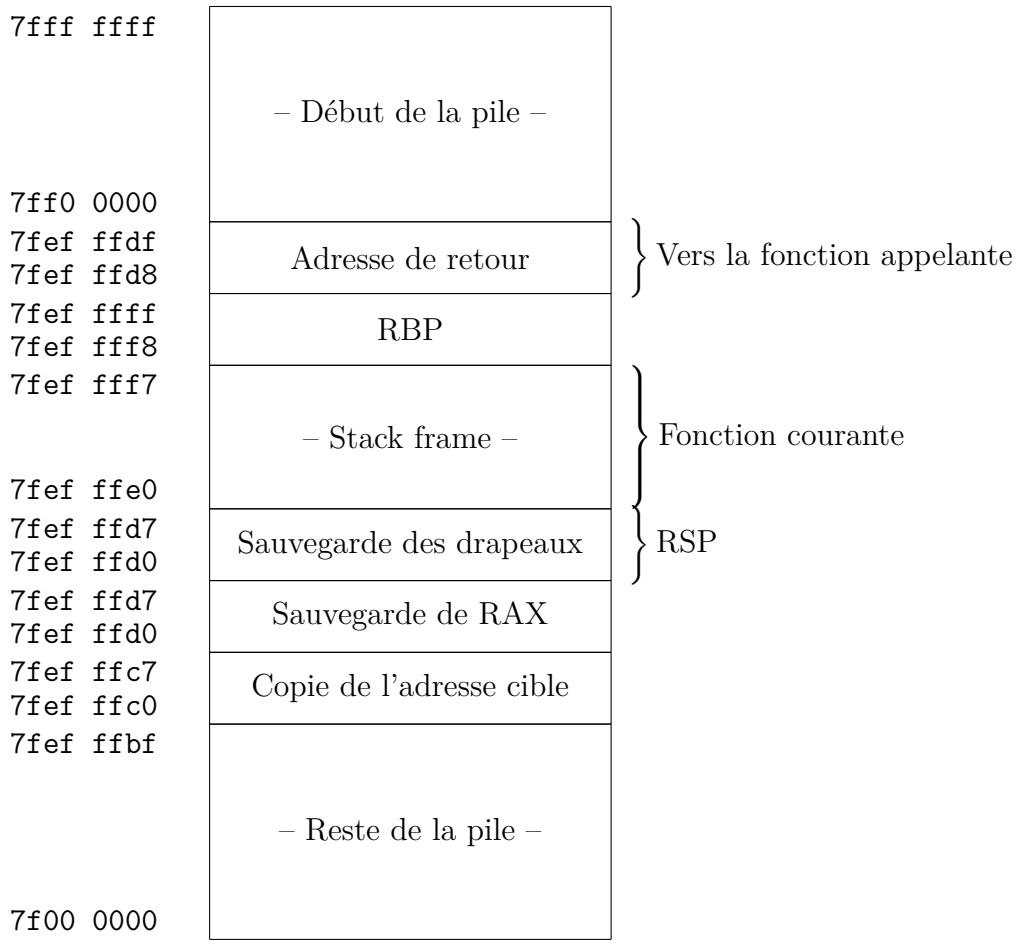

FIGURE 2.5 – Configuration de la pile juste avant de sortir de la table des sauts

quitter la table des sauts est décrite dans la figure 2.5.

Par ailleurs, il est possible que les drapeaux (*flags*), modifiés par certaines instructions dont `test`, `cmp` ou `sub`, soient utilisés après un saut. Malheureusement, les comparaisons effectuées par les tobbogans modifient ces drapeaux, modifiant alors la sémantique initiale du programme. Par conséquent il est nécessaire de sauvegarder ces drapeaux avant d'entrer dans la table et de les restaurer avant d'en sortir. Le jeu d'instruction x86-64 propose les instructions `pushf` et `popf` permettant respectivement de pousser et retirer les valeurs de ces drapeaux sur la pile. Puisque ces instructions manipulent la pile, l'adresse pointée par `rsp - 8` sert à stocker la valeur des drapeaux, d'où l'intérêt de laisser cet emplacement disponible et sauvegarder le registre de travail plus loin. En conséquence, le code d'un cas d'un tobbogan d'une table de délégation des sauts d'une fonction ressemble finalement au

```
1  _this_case:
2    cmp eax, _bb1_start
3    jl _next_case
4    cmp eax, _bb1_end
5    jge _next_case
6    add rax, decalage
7    popf
8    mov QWORD PTR [rsp - 0x18], rax
9    mov rax, QWORD PTR [rsp - 0x10]
10   jmp QWORD PTR [rsp - 0x18]
11 _next_case:
12   ;; ...
```

Listing 21: Code d'un cas du tobbogan d'une table de délégation des sauts indirects d'une fonction

code du listing 21.

Puisque le registre de travail est modifié dans le prologue de la table de délégation, il est nécessaire d'effectuer la sauvegarde de `rax` avant d'y entrer. Il s'agit par ailleurs d'un bon moment pour sauvegarder les drapeaux. En conséquence, le code du saut inconditionnel est remplacé par le code du listing 22.

```
1 mov QWORD PTR [rsp - 0x10], rax
2 pushf
3 jmp _dispatch_table
```

Listing 22: Code remplaçant l'instruction de saut originelle

D'après l'hypothèse H3, les sauts inconditionnels peuvent aussi être la résultante d'une optimisation d'un appel récursif terminal. Cela veut dire que le saut est en fait un appel de fonction n'allouant pas d'espace supplémentaire sur la pile et ne nécessitant pas de retour supplémentaire. Autrement dit, un seul retour de fonction permet de dépiler tous les appels récursifs terminaux en une seule fois. Cependant, la construction des tables de délégation des sauts indirects n'en tient pas compte. En effet, les cibles identifiées par ces tables sont les instructions appartenant à la même fonction que l'instruction de saut.

Afin de permettre au processus cible de rediriger les appels récursifs terminaux, deux solutions sont possibles : rajouter l'adresse de début de toutes les fonctions du programme dans toutes les tables de délégation des sauts ou rediriger l'exécution vers la table des

appels de fonction en cas d'erreur. Nous avons opté pour la seconde approche, pour deux raisons. La première est que recopier l'intégralité du tobbogan de la table des appels dans chacune des tables de saut représente une perte de mémoire conséquente qu'il est possible d'éviter en redirigeant simplement le flot de contrôle vers la table des appels. La seconde raison est que la table des appels de fonction est capable de gérer tous les cas aux limites présentés dans la sous-section 2.3.4.

Par conséquent, la fin des tables de saut ne comporte par un code de traitement des erreurs mais une redirection vers la table des appels de fonction. Par ailleurs, puisque les tables de saut et la table des appels utilisent le même registre de travail, il est possible de rediriger le flot de contrôle directement vers le tobbogan de la table des appels, sans passer par le moindre prologue. Ainsi, la fin des tables de saut ressemble au code du listing 23.

```
1 _error_code:
2   jmp _call_dispatch_table_sled
```

Listing 23: Code d'erreur à la fin des tables de délégation des sauts inconditionnels

2.3.4 Les appels et retours de fonctions contournant la PLT

Les tables présentées à la sous-section précédente permettent de rediriger la majorité des branchements indirects présents dans le programme cible. Néanmoins, il existe quelques cas particuliers qui nécessitent de complexifier encore le mécanisme des tables de délégation. En effet, les tables montrées jusqu'à présent ne permettent de gérer que les cas suivants :

— Un appel de fonction vers le binaire instrumenté ou sa PLT.
— Un retour de fonction du binaire instrumenté vers lui-même.
— Un saut d'une fonction dans elle-même.

Dans cette sous-section, nous nous intéressons aux cas aux limites qui échappent à la logique des tables de délégation présentées précédemment et nous expliquons les solutions trouvées pour rediriger correctement les branchements indirects concernés. Afin de rendre notre propos le plus clair possible, nous présenterons ces cas aux limites sous la forme d'un exemple concret mettant en scène une bibliothèque partagée et un binaire principal.

Une bibliothèque et des pointeurs de fonction

Nous supposons l'existence d'une bibliothèque partagée définissant le type de donnée `Data`, tel que décrit dans le listing 24.

```
1  typedef struct Data Data;
2  struct Data {
3    int value;
4    void (*method)(Data* self);
5  };
```

Listing 24: Définition du type de donnée `Data`

Ce type de donnée est simple et contient une valeur ainsi qu'une méthode associée. Cette méthode prend un pointeur vers un objet `Data` en paramètre, il s'agit de l'équivalent de `this` en C++. En effet, le langage C n'a pas de construction du langage permettant la programmation orientée objet. Par conséquent, il est nécessaire de faire apparaître explicitement dans le code tous les rouages internes de ce paradigme de programmation.

On peut supposer par ailleurs l'existence de deux fonctions : une implémentation de la méthode `method` du type `Data` ainsi qu'un constructeur permettant de créer un objet de type `Data`. Nous considérons alors le code du listing 25.

Bien qu'il soit très simple en apparence, ce code est déjà très intéressant pour notre étude. En effet à la ligne 8, l'attribut `method` de l'objet `data` est initialisé avec l'adresse de la fonction `impl_method`. Grâce à cela, il serait possible d'appeler cette méthode en écrivant `data.method(&data)`, la fonction `impl_method` serait appelée. Cette affectation

```
1  void impl_method(Data* self) {
2    printf("Value: %d\n", self->value);
3  }
4
5  Data new_data(int value) {
6    Data data;
7    data.value = value;
8    data.method = impl_method;
9    return data;
10 }
```

Listing 25: Implémentation d'un constructeur pour `Data` et de sa méthode `method`

est compilée en un code assembleur ressemblant au code du listing 26.

```
1 mov rax, QWORD PTR [rip+0x2ea2] ; 3fe8 <impl_method@@Base+0x2edf>
2 mov QWORD PTR [rbp-0x10], rax
```

Listing 26: Code assembleur correspondant à `data.method = impl_method`

Puisque notre bibliothèque a pour objectif d'être chargée dynamiquement, il est nécessaire de compiler le code de manière à le rendre indépendant de sa position dans l'espace d'adressage (*position-independent code*). Par défaut, `gcc` remplace donc l'adresse même de `impl_method` par l'entrée correspondante dans la section `.rela.dyn`[9] de la bibliothèque. Ce qui est intéressant, c'est que l'adresse écrite dans `rax` puis dans l'emplacement mémoire correspondant à l'attribut `method` de notre objet (ici `rbp-0x10`) soit une adresse dans la bibliothèque elle-même. C'est à dire que peu importe si la fonction `new_data` est appelée par notre bibliothèque elle-même, le binaire principal du programme qui l'utilisera ou une autre bibliothèque, ce pointeur ciblera toujours la première instruction de la fonction `impl_method` dans dans notre bibliothèque.

Un programme principal

On suppose à présent le programme du listing 27, utilisant notre bibliothèque. L'appel de la méthode `method` à la ligne 5 peut être compilé en un code assembleur similaire au code du listing 28.

Dans un premier temps, la valeur de l'attribut `method` de notre objet est écrit dans `rdx`. Il s'agit de l'adresse de `impl_method` comme expliqué plus tôt. Puis, le premier

9. Cette section contient les informations de rebasement de toutes les sections de la bibliothèque à l'exception de la PLT. Elle joue un rôle similaire à cette dernière et permet d'avoir du code indépendant de sa position dans la bibliothèque.

```
1 #include <ourlib.h>
2
3 int main(void) {
4   Data data = new_data(42);
5   data.method(&data);
6   return 0;
7 }
```

Listing 27: Programme utilisant la bibliothèque

```
1 mov    rdx, QWORD PTR [rbp-0x18]
2 lea    rax, [rbp-0x20]
3 mov    rdi, rax
4 call   rdx
```

Listing 28: Appel à la méthode `method` de `data`

paramètre de la fonction est initialisé : l'adresse de notre objet est écrit dans `rdi` en utilisant `rax` comme intermédiaire. Et enfin, la fonction est appelée de manière indirecte en utilisant l'adresse écrite précédemment dans `rdx`. Si ce programme est exécuté de manière conventionnelle, il n'y aura aucun souci et le processus affichera la valeur de `data.value` comme prévu. Toutefois, si l'exécution du processus est instrumentée par Damas, une erreur se produirait.

Damas et les appels vers l'étranger

Le scénario développé précédemment pose un grave problème à Damas. En effet, d'après l'hypothèse H1, la première instruction de toutes les fonctions du programme sont considérées valides pour un appel de fonction indirect. Dans le paragraphe dédié à la table de délégation des appels de fonctions indirects, nous avons expliqué que les fonctions considérées étaient les fonctions du binaire principal ainsi que les entrées de la PLT, partant du principe que le flot de contrôle du processus exécuterait le code des bibliothèques uniquement en y entrant via la PLT. Or ici, l'instruction `call rdx` cible une adresse se trouvant directement dans l'espace d'adressage réservé à la bibliothèque, contournant complètement la PLT.

Dans de telles circonstances, la table des appels de fonctions ne peut pas considérer l'adresse cible comme valide et donc l'exécution du programme échoue sur un faux positif, croyant être la cible d'une attaque. L'objectif ici est de permettre au processus cible d'effectuer des appels de fonctions directement dans l'espace d'adressage de bibliothèques auxquelles il est lié sans passer par la PLT et en s'assurant de la validité du branchement. En effet, il serait naïf de penser que tout branchement dans l'espace d'adressage d'une bibliothèque est valide. Il s'agit même en réalité d'un comportement plutôt alarmant [10]. L'astuce consiste donc en l'identification des adresses cibles valides au sein des

10. La bibliothèque standard du C est réputée pour être une cible de choix dans le cas d'attaques par détournement du flot de contrôle. En effet, cette bibliothèque contient un ensemble important de gadgets permettant des attaques ROP élaborées ainsi que l'interface de nombreux appels systèmes intéressants tels que `system` ou `execv`. Ainsi, un branchement directement dans l'espace d'adressage de la bibliothèque

bibliothèques chargées en mémoire.

Afin de parvenir à valider ou non un appel de fonction indirect en dehors des cibles définies dans la table des appels de fonctions, nous utilisons les symboles fournis par les fichiers ELF correspondant aux bibliothèques. En effet, puisqu'il est question d'appeler des fonctions dans ces bibliothèques, il est légitime de penser que l'adresse cible de l'appel correspond à l'adresse de début d'une fonction. Or, les bibliothèques exposent leurs fonctions au moyen des symboles ELF. C'est la raison pour laquelle nous utilisons ces symboles afin de faire correspondre l'adresse cible du branchement et l'adresse à laquelle une fonction est censée être chargée.

Pour mener à bien cette tâche, nous avons modifié le code d'erreur de la table des appels de fonctions afin de le préfixer d'une instruction trap. Ainsi, lorsqu'un appel de fonction suspect est détecté, le processus est mis en pause et Damas vérifie manuellement la validité du branchement. Pour ce faire, notre outil récupère l'adresse cible du branchement et cherche la bibliothèque dans laquelle l'adresse se trouve. S'il n'y en a pas, il s'agit manisfestement d'une erreur et donc Damas laisse le processus cible continuer son exécution dans le code de gestion des erreurs de la table des appels de fonctions. Si au contraire, la bibliothèque peut être identifiée, ses symboles sont testés afin de savoir si l'un d'entre eux est chargé à l'adresse discriminée par notre instruction de branchement. Si c'est le cas, l'exécution du processus cible est redirigé vers cette adresse en modifiant directement ses registres, économisant un saut une fois l'exécution relancée. Sinon, l'exécution est relancée telle quelle, exécutant le code de gestion des erreurs.

La raison pour laquelle ces vérifications sont effectuées paresseusement par Damas pendant l'exécution du processus cible est qu'il n'est pas toujours possible d'obtenir les informations nécessaire à la gestion de ce type d'appels « vers l'étranger » au moment de l'instrumentation du processus. En effet, au moment de l'exécution du processus cible où Damas s'attache et procède à l'instrumentation du code, toutes les bibliothèques ne sont peut-être pas encore chargées [11]. Ainsi, il n'est pas possible de calculer les adresses auxquelles les symboles de ces bibliothèques seront eux aussi chargés. Il est donc nécessaire d'attendre qu'un de ces branchements vers l'étranger se produise pour être certain que la bibliothèque correspondante soit chargée et que les vérifications idoines soient possibles.

C peut être considéré comme un comportement très suspect.

11. Il est possible de lancer le processus cible directement sous l'égide de Damas de manière à ce que toute son exécution soit protégé par notre outil (via une combinaison classique de `fork` et `execv`). C'est la raison pour laquelle nous prenons des précautions sur le chargement des bibliothèques.

Retour vers l'étranger

De la même manière qu'une instruction `call` peut cibler l'adresse d'une fonction dans l'espace d'adressage d'une bibliothèque en contournant complètement la PLT, il est possible qu'une fonction appartenant au binaire principal retourne vers du code appartenant à une bibliothèque. C'est typiquement ce qui se produit quand une fonction d'une bibliothèque prend un pointeur de fonction en paramètre. Par exemple, en C++, la fonction `sort` de la bibliothèque standard peut prendre une fonction binaire de comparaison comme troisième paramètre. Si le programme principal définit une telle fonction et la donne en paramètre à `sort`, celle-ci l'appellera durant son exécution. Ainsi, l'adresse de retour de cette fonction de comparaison ciblera une instruction dans `sort` et donc dans la bibliothèque standard de C++.

Le problème ici est analogue au cas précédent, à savoir que la table des retours de fonction n'est pas en mesure de contenir l'adresse de retour de la fonction. La solution proposée est similaire à la précédente. Nous vérifions dynamiquement si l'adresse cible du retour de fonction correspond à l'instruction placée juste après une instruction `call` dans le code de la bibliothèque.

2.4 Optimisation des tables de délégation

La redirection de branchements indirects via les tables de délégation est procédé astucieux. Il nous permet de contourner le problème que pose ces branchements, à savoir que leur cible n'est connu qu'au moment d'exécuter l'instruction de branchement. Néanmoins le défaut majeur de cette approche est son impact sur les performances. En effet, les tables de délégation transforment un simple branchement en une recherche dans une structure de données.

Les tables de délégations présentées précédemment prennent la forme d'un tobbogan dont les cas sont testées l'une après l'autre, il s'agit d'une recherche linéaire sur les cas du tobbogan. Ceci implique que les premières cas du tobbogan sont atteintes plus rapidement que les dernières cas. Comme présenté plus en détail dans la section 3.2, les tables de délégation des appels et des retours de fonctions contiennent un très grand nombre de cas et sont les tables les plus utilisées par les processus protégés par Damas. Par conséquent l'ordre d'apparition des adresses cibles potentielles d'un branchement indirect dans la table de délégation idoine est d'une importance capitale.

Par défaut, notre approche construit les tables de délégations sans considération pour

les performances du processus. En effet, les cas sont placées par ordre croissant de l'adresse cible [12]. Il est néanmoins possible d'améliorer notre approche en représentant les tables de délégation autrement. Dans cette section nous présentons les deux optimisations que nous avons implémenté dans Damas afin de réduire grandement l'impact des tables de délégation sur les performances des processus cibles.

2.4.1 Un tri par utilisation décroissante

Les cas d'une table de délégation sont visitées linéairement, imposant donc un biais en faveur des premiers cas de la table. Or, tous les branchements indirects d'un processus ne peuvent pas cibler tous les cas de leur table de délégation et a fortiori ils ne les ciblent pas uniformément. Par conséquent, certains cas sont bien plus utilisés que d'autres et certains ne le sont absolument jamais [13].

Par conséquent, il serait intéressant de trier les cas d'une table de délégation de manière à placer en premier les cas les plus utilisés. Cette réorganisation de la table de délégation permettrait de réduire drastiquement le temps passé à traverser la table pour les cas les plus utilisés. Une tellement optimisation serait particulièrement adaptée à des tables dont la majorité des cas n'est jamais ou très peu utlisée mais dont certains cas sont clairement favorisés. C'est typiquement le cas de la table des appels de fonctions. En effet, cette table contient un cas par fonction dans le binaire principal. Or la majorité de ces fonctions n'est jamais appelée indirectement.

Cette organisation des tables de délégation impose néanmoins de connaître à l'avance les cas les plus utilisés afin de les ordonner correctement. Pour cela, il est possible, dans un premier temps d'utiliser des traces d'une précédente exécution du programme sous la protection de Damas. En effet, il est possible de munir les tables de délégation de compteurs permettant de connaitre exactement le nombre d'utilisation de chaque cas. Ce faisant, il est possible de savoir quels cas sont les plus succeptibles d'être utilisés à nouveau d'une exécution à l'autre, permettant alors d'ordonner correctement les cas des tables de délégations pour les exécutions ultérieures.

Toutefois, cette façon de procédé nécessite plusieurs exécutions du programme pour être mise en place. Afin de bénéficier de cette optimisation sans avoir besoin de relancer le programme, il est possible de procéder autrement. Puisque Damas reste attaché au

12. Cette approche a été choisie pour sa simplicité de mise en place et faciliter le débogage de Damas.
13. C'est typiquement le cas des fonctions qui ne sont jamais appelées indirectement par le programme. Un cas existe dans la table des appels de fonctions, mais il n'est jamais utilisé.

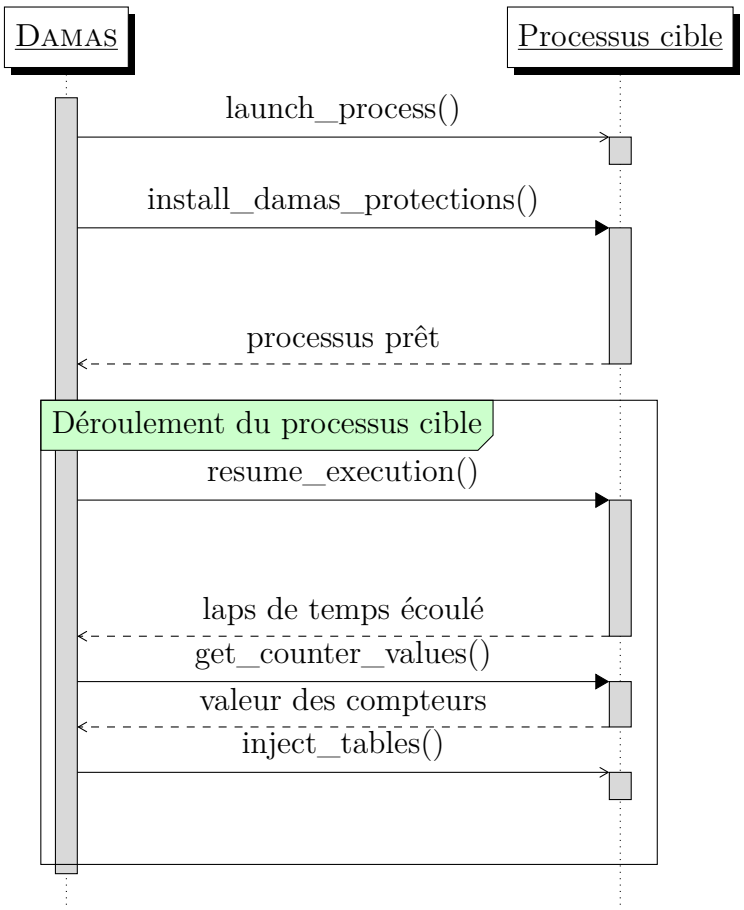

Figure 2.6 – Modification dynamique des tables de délégation par Damas

processus cible durant son exécution, il est possible de le faire intervenir régulièrement dans le but de réordonner les tables de délégation dynamiquement. Damas pourrait utiliser les valeurs actuelles des compteurs pour dynamiquement réordonner la table de manière à toujours avoir les cas les plus utilisés en premier. La figure 2.6 donne un aperçu du fonctionnement de ce procédé.

2.4.2 Une représentation en arbre binaire

Le tri des cas d'une table de délégation a pour avantage de réduire le temps passé à traverser la table pour atteindre les cas les plus utilisés, réduisant par conséquent le temps total passé à traverser la table durant l'exécution du processus. Bien que cette optimisation soit particulièrement adaptée aux tables dont l'utilisation des cas est très inégale, elle perd beaucoup de son intérêt quand beaucoup de cas de la table sont fréquemment utilisés.

Pour ce genre de tables, il serait plus intéressant de changer toute la représentation de la table en mémoire. Plutôt que traverser la table linéairement, utiliser une représentation en arbre binaire serait plus approprié. De plus, cette optimisation ne nécéssite pas l'utilisation des compteurs dans le but d'ordonner la table. Il est donc possible d'utiliser la représentation des tables en arbre dès le début de l'exécution du processus cible sous la protection de Damas.

Chapitre 3

Évaluation

Dans l'objectif de nous assurer du bon fonctionnement de notre approche en termes de sécurité ainsi que de mesurer l'impact de notre solution sur les performances du processus cible, nous avons mené diverses expérimentations. D'abord nous devons évaluer la capacité de notre outil à détecter des attaques et les arrêter ainsi que définir les limites de notre approche, c'est-à-dire les faiblesses de celles-ci qui permettent de la contourner pour réussir à attaquer le processus cible. Puis, nous devons évaluer le surcoût en performances de notre solution sur le processus cible, dans le but de nous comparer avec les solutions existantes. Dans ce chapitre, nous présentons nos protocoles expérimentaux ainsi que les résultats de nos expériences.

3.1 Évaluation de sécurité

Nos travaux étant axés autour de la protection contre les attaques par corruption de mémoire, il est nécessaire de nous assurer que notre solution remplisse effectivement sa mission. Par conséquent nous avons mis en place des scénarios d'attaques sur des petits programmes de test afin de comparer le déroulement des attaques selon si le processus est protégé par Damas ou non.

Nous avons développé deux programmes de test. Le premier appelle une fonction qui écrit « hello » sur la sortie standard en utilisant un pointeur de fonction. Le second appelle la même fonction puis en appelle une seconde qui termine le programme avec le code 0x99, en utilisant l'appel système `exit`.

Nous considérons un modèle d'attaque classique dans lequel l'attaquant a pu écrire où il voulait en mémoire, sur la pile comme dans le tas, avant que le processus n'exécute l'instruction de transfert de contrôle détournée par l'attaquant. Afin de simplifier la mise en place de nos tests, nous n'avons pas utilisé d'entrée telle que l'entrée standard, `stdin`, par laquelle l'attaquant peut corrompre le contenu de la mémoire en utilisant une fonction vulnérable, comme la fonction `gets` de la bibliothèque C. Nous avons au contraire

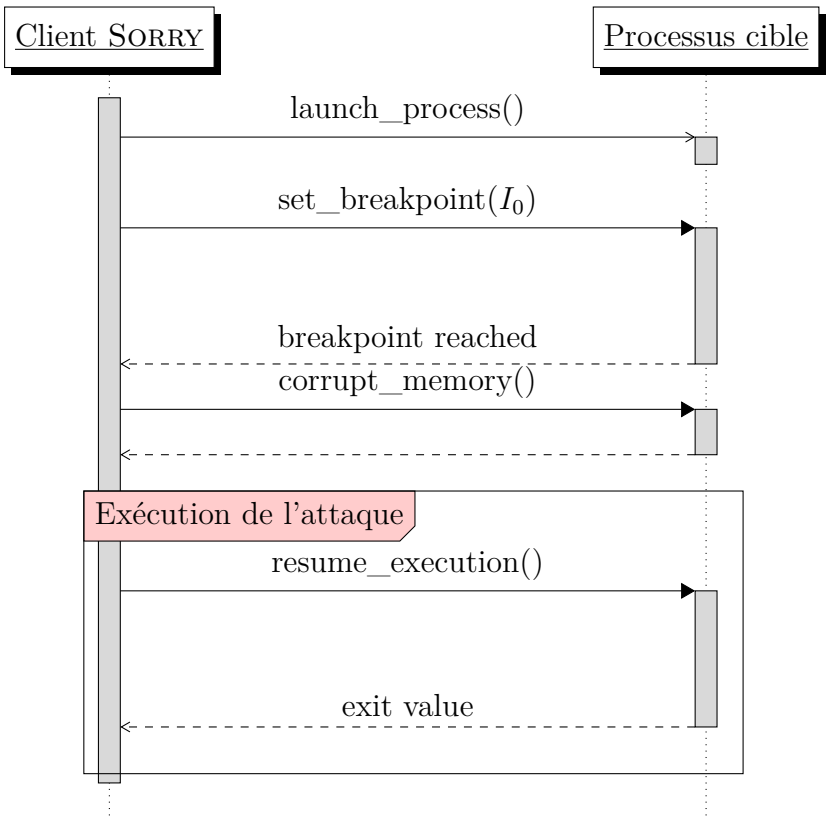

Figure 3.1 – Déroulement des scénarios de test de sécurité. I_0 désigne l'instruction de branchement indirect vulnérable utilisée pour détourner le processus de son flot de contrôle.

instrumenté le processus vulnérable grâce à notre bibliothèque Sorry afin de corrompre la mémoire de manière ad-hoc juste avant l'exécution du branchement indirect ciblé et observer le comportement du processus cible, grâce aux fonctionnalités de notre bibliothèque. Notons que ce choix n'impacte pas notre mécanisme de protection qui ne protège pas le programme contre les modifications des données mais assure l'intégrité de son flot de contrôle. Cette protection s'applique, quelque soit le moyen utilisé par l'attaquant pour modifier les données de flot de contrôle. Le déroulement de ces scénarios de test est détaillé dans la figure 3.1.

Nous avons tenté d'attaquer nos programmes de test de plusieurs manières. Nous avons défini les scénarios suivants :

— Le programme appelle la fonction `exit` de la bibliothèque C mais après avoir sauté les premières instructions gérant l'état de la pile (de façon à sauter à une adresse invalide du point de vue de Damas) ;

Scénario	Non-protégé	Protégé
appel à `exit`	✓	✗
ROP classique	✓	✗
Control-Flow Bending	✓	✓

TABLE 3.1 – Résultats des tests de sécurité, un ✓ signifie que l'attaque s'est déroulée normalement et un ✗ indique que DAMAS a bloqué l'attaque.

— Le programme exécute une attaque ROP classique sans chercher à se cacher de DAMAS ;
— Le programme exécute une attaque proche de *Control-Flow Bending* (CARLINI et al. 2015) que DAMAS ne peut pas détecter.

Dans chaque scénario, l'objectif de l'attaque est de faire exécuter un appel système `exit` par le processus avec un code d'erreur précis de manière à savoir facilement si l'attaque a été possible ou non. Chacun de ces scénarios est joué sur un processus non-protégé et sur un processus protégé dans l'objectif de comparer le résultat. La table 3.1 répertorie le déroulement de ces scénarios.

Afin de discuter plus avant de ces résultats, il est important de rappeler le fonctionnement de DAMAS. Notre outil est une solution de CFI implémentant une CDI basée sur le code binaire, capable de s'attacher à un processus en cours d'exécution pour le protéger dynamiquement. Par conséquent, puisqu'il est particulièrement difficile d'obtenir un CFG complet de manière fiable depuis le code binaire, DAMAS ne cherche pas à se baser sur un modèle précis de CFG. Nous utilisons seulement notre désassemblage du binaire qui identifie toutes les fonctions et blocs de bases légitimes mais ne cherche pas à déterminer statiquement le CFG. Bien que cette approche nous donne une plus grande assurance que le programme instrumenté continuera de fonctionner correctement, DAMAS repose sur une surapproximation importante du CFG. En effet, nous n'avons défini que quelques classes d'équivalence : celle des appels de fonctions, celle des retours de fonction et celle des sauts au sein d'une même fonction. Par conséquent, il est possible de comparer DAMAS à un CFI à très gros grain du point de vue de la sécurité.

Ainsi, les deux scénarios qu'un CFI est censé pouvoir gérer, à savoir l'appel à `exit` à quelques octets près ainsi que le ROP classique, sont gérés également par DAMAS. Dans le premier scénario, le processus tente de sauter à une adresse qui ne correspond à aucun symbole de la bibliothèque C, donc cette adresse n'apparaît nulle part dans la table des appels de fonctions que DAMAS a générée pour le processus, c'est pourquoi l'attaque échoue.

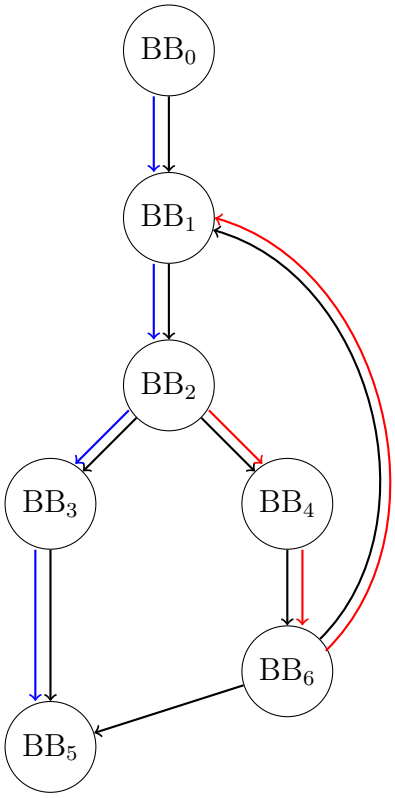

FIGURE 3.2 – Illustration d'une attaque type Control-Flow Bending sur un CFG. Le chemin en bleu correspond au flot de contrôle voulu du processus et le chemin en rouge correspond au chemin pris à cause d'une attaque.

Le second scénario consiste à assembler une poignée de gadgets de manière à pousser le processus cible à exécuter un appel système `exit`. La particularité de ce scénario est que les adresses auxquelles le processus essaie de retourner ne sont pas des adresses de retour valides. En effet, la pile est corrompue de manière à retourner aux adresses des gadgets choisis pour l'attaque. Toutefois, ces gadgets ne sont pas nécessairement placés juste après une instruction `call`, critère pourtant nécessaire pour que DAMAS considère une adresse de retour comme valide. Par conséquent, lorsque le processus exécute la première instruction `ret` de l'attaque, l'adresse ciblée n'apparaît pas dans la table des retours de fonction de DAMAS et l'attaque échoue.

Le troisième scénario est plus intéressant. En effet, il s'agit d'un scénario qu'un CFI classique ne pourrait pas détecter. Plus précisément nous avons mis en place une attaque qui s'apparente au *control-flow bending* (CARLINI et al. 2015), une attaque qui détourne le flot de contrôle du processus cible tout en respectant les contraintes imposées par le CFG.

L'idée sous-jacente est que la grande faiblesse du CFI est sa non-sensibilité au contexte. CDI souffre également de cette limitation. Ainsi, cette attaque cherche à détourner le processus tout en restant dans le CFG. Pour être plus précise, si un appel de fonction peut cibler deux fonctions différentes, l'attaque peut pousser le processus à appeler une fonction alors que le contexte d'exécution devrait l'amener à appeler l'autre. La figure 3.2 donne un exemple de CFG détourné sans qu'un CFI ne puisse le détecter. À l'échelle de tout un programme, cette faiblesse permet d'exécuter du code totalement arbitraire comme Carlini et al. 2015 le décrit. Notre scénario prend place dans le programme de test suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void print_message(void) {
5    puts("Hello");
6  }
7
8  void exit_0x99(void) {
9    exit(0x99);
10 }
11
12 int main(void) {
13   print_message();
14   exit_0x99();
15   return 0;
16 }
```

Ce programme affiche « Hello » dans la console puis appelle la fonction `exit_0x99` qui termine le programme avec comme valeur de retour 0x99. Étant donné que Damas fait peu d'hypothèses sur le CFG originel du programme, il est possible ici de le duper [1] en faisant retourner `print_message` non pas juste avant l'appel à `exit_0x99` (son adresse de retour initiale) mais juste après cet appel (donc, l'adresse de retour de `exit_0x99`). L'appel à cette fonction n'est alors jamais effectué et le programme se termine avec une

1. Nous rappelons que la mémoire est corrompue avec un client Sorry prévu à cet effet, ce qui nous facilite la mise en place d'attaques. Ainsi nous pouvons détourner le processus cible, bien qu'aucune entrée vulnérable n'apparaisse dans ce bout de code.

valeur de retour de 0 au lieu de 0x99. Notre expérimentation montre qu'une telle attaque se déroule avec succès sur un processus non-protégé mais aussi sur un processus protégé par DAMAS, comme expliqué ci-dessus.

3.2 Évaluation des performances

L'adoption d'une solution de cyber-sécurité est en partie conditionnée par l'impact de celle-ci sur les performances du système qu'elle protège. Cela est particulièrement le cas en sécurité applicative. SZEKERES et al. 2013 montre que les seules protections déployées par défaut par les compilateurs ou sur les systèmes d'exploitation sont celles dont l'impact sur les performances est négligeable. Typiquement, la seule protection de type CFI présentée dans l'article et qui est déployée par défaut dans les compilateurs tels que GCC ou LLVM, les *stack cookies*, a un surcoût en temps d'exécution maximal de 5%. Pour cette raison, nous avons évalué l'impact de DAMAS sur les performances des programmes qu'il protège.

3.2.1 Protocole expérimental

Nous avons procédé à une évaluation des performances offertes par DAMAS sur trois ensembles de programmes. Tout d'abord nous avons constitué un petit corpus de programmes ayant une utilisation intensive du *Central Processing Unit* (CPU). Ce corpus contient les logiciels de compression GZIP 1.10 et BZIP2 1.0.8, le gestionnaire de bases de données SQLITE3 ainsi que le compilateur C TCC 0.9.27.

Puis, nous avons constitué un autre corpus contenant des applications serveurs plus typiques des applications que DAMAS serait susceptible de protéger. Ce corpus nous permet d'apprécier un impact de DAMAS sur les performances plus réaliste vis à vis du cas d'utilisation voulu de notre outil que sur des programmes avec une utilisation intensive du CPU. Il contient le serveur HTTP NGINX 1.18, compilé normalement et compilé statiquement à l'exception de la bibliothèque C (ce programme est appelé NGINX-STATIC) et le serveur MQTT MOSQUITTO 1.6.9.

Le troisième corpus contient un sous ensemble de la suite de benchmarks SPEC CPU 2006. Il s'agit des programmes C qu'il a été possible de protéger avec DAMAS sans que le temps d'exécution n'explose [2]. Par conséquent, ce corpus est constitué de ASTAR, BZIP2,

2. Sans optimisation, DAMAS impose un surcoût en temps d'exécution important sur ces programmes et certains programmes de SPEC CPU 2006 sont plus particulièrement impactés. Par exemple, le temps d'exécution de GCC avoisinait une semaine.

GOBMK, HMMER, H264REF, LIBQUANTUM, MCF, SJENG et SPECRAND. Afin de nous assurer de la fiabilité de nos résultats, nous avons mesuré les performances de ces programmes en utilisant autant de configurations proposées par SPEC CPU 2006 que possible [3]. Par conséquent, les résultats présentés dans ce manuscript tiennent compte des différentes configurations données au programme cible. Notre choix d'inclure ou non une configuration dépend entièrement du temps nécessaire à l'exécution du programme cible [4]. En effet, certains programmes couplés à certaines configurations étaient trop longs à exécuter pour la conduite de ces tests en des temps raisonnables.

Nous avons testé DAMAS sur ces corpus de programmes sur une plateforme de test avec un CPU Intel Xeon E5-1603 v4 @ 2.80 GHz executant un système Linux Fedora 34 avec une noyau Linux 5.11.15-300.fc34.x86_64. Pour les deux premiers corpus, chaque programme était contenu dans un conteneur Docker dédié avec une copie de DAMAS ainsi que toutes les dépendences nécessaires à l'exécution de l'expérience (comme la commande `time` ou le logiciel APACHEBENCH). Pour le corpus de SPEC 2006, tous les programmes étaient placés dans un même conteneur avec une copie de DAMAS.

Tous ces programmes ont été exécutés dans quatre configurations :

— *référence* : le programme est exécuté sans protection pour servir de référence ;
— *non-triée* : le programme est protégé par DAMAS sans optimisation ;
— *trié* : le programme est protégé par DAMAS et les entrées des tables sont triées [5] ;
— *arbre* : le programme est protégé par DAMAS et les tables utilisent une représentation en arbres binaires.

L'intérêt de ces quatres configurations est de comparer les performances d'un processus protégé par rapport à un processus normal aussi que de donner un aperçu du gain de performances obtenu grâce à une optimisation simple comme un changement de représentation des tables de délégation.

Afin de mesurer ces temps d'exécution, nous n'avons pas greffé DAMAS sur des processus existants, mais avons demandé à notre outil de lancer les processus lui-même [6]. Ainsi,

3. SPEC CPU 2006 fournit un ensemble de programmes et de jeux de données sous formes de fichiers de données. De plus, cette suite de benchmarks décrit des scénarios d'exécution des programmes en leur fournissant des paramètres et une entrée standard permettant de tester ces programmes avec différentes configurations. Dans ce manuscript, ce sont ces scénarios qui sont appelés « configurations ».

4. Certains programmes comme H264REF ont un temps d'exécution normalement assez long. Leur exécution sous l'égide de DAMAS sans optimisation devenait si longue qu'il n'aurait pas été possible d'obtenir assez de résultats pour l'entièreté du corpus de programmes en deux mois d'évaluation.

5. L'exécution *non-triée* génère des traces d'exécution permettant de trier les tables de l'exécution *triée* suivante.

6. le lancement d'un processus par DAMAS se fait via une combinaison classique de `fork` et `execv`.

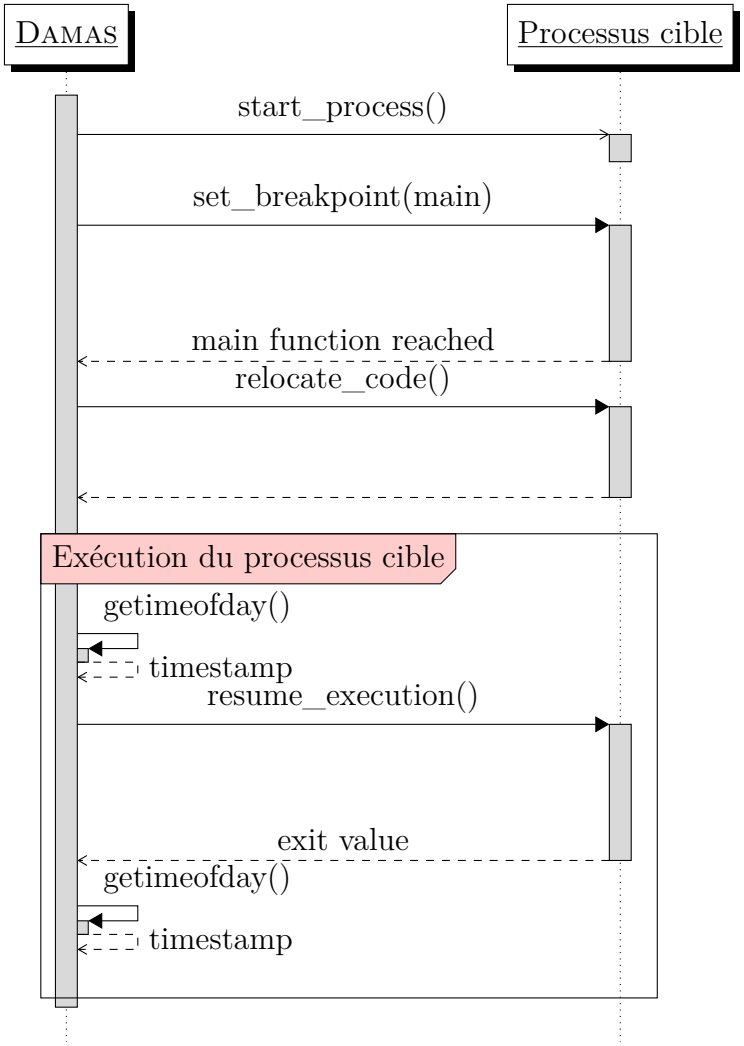

Figure 3.3 – Déroulement d'une exécution de Damas dans le cadre des tests de performances

la mesure du temps d'exécution du processus cible commence à partir du moment où le code est entièrement rebasé et se termine avec la terminaison du processus. La figure 3.3 décrit précisément le déroulement d'un test de performances de DAMAS. La mesure commence alors que le processus a déjà atteint la fonction `main`, ce qui exclut le chargement de la bibliothèque C ainsi que toutes initialisations faites avant l'exécution de la fonction `main`. Ceci est le résultat d'une contrainte technique liée au fonctionnement de notre bibliothèque, SORRY. En effet, afin de pouvoir en utiliser toutes les fonctionnalités, il est nécessaire que la bibliothèque C soit chargée en mémoire. Or, le moyen le plus simple de s'en assurer est de placer un *breakpoint* à l'entrée de la fonction `main` et de laisser le processus atteindre ce *breakpoint*. Par conséquent, le temps d'exécution mesuré correspond à l'exécution de la fonction `main`. Néanmoins, le temps d'exécution du code préalable à la fonction `main` étant négligeable en l'absence de DAMAS, nous n'avons pas utilisé de mécanisme similaire pour l'exécution du processus en mode *référence* dont l'exécution est mesurée de son point d'entrée jusqu'à sa terminaison.

3.2.2 Résultats

Nous avons publié les résultats de l'évaluation des performances de DAMAS sur les deux premiers corpus de programmes dans LE BON et al. 2021. Dans cet article, nous avons dans un premier temps présenté une évaluation de DAMAS sur un corpus de programmes avec une utilisation intensive du CPU, dont les temps d'exécution moyens sont présentés dans la table 3.2. Cette évaluation nous permettait de nous comparer avec d'autres approches de CFI dont l'évaluation des performances se fait essentiellement sur ce genre de programmes (MASHTIZADEH et al. 2015 ; NIU et TAN 2015 ; SZEKERES et al. 2013 ; VAN DER VEEN et al. 2015).

Par ailleurs, l'article présente également une évaluation des performances sur un second corpus de programmes dont la particularité est d'être représentatif du type de programmes que DAMAS pourrait protéger : des serveurs, dont la durée d'exécution est virtuellement infinie et qu'on pourrait ne pas vouloir arrêter pour y ajouter des protections. La table 3.3 présente les résultats concernant ce corpus.

Les tables présentées dans cette sous-section montrent le temps d'exécution moyen mesuré pour chaque programme dans les quatre configurations décrites à la sous-section précédente. Ces temps moyens sont des moyennes arithmétiques des temps mesurés. Les surcoûts supérieurs à 100% sont exprimés par une multiplication du temps d'exécution, par exemple un surcoût de 100% est écrit $\times 2$. Les surcoûts en temps d'exécution inférieurs

Programme	Référence	Damas (non-triée)		Damas (arbre)		Damas (triée)	
	temps (s)	temps (s)	surcoût	temps (s)	surcoût	temps (s)	surcoût
BZIP2	198.86	935.24	×3.7	226.70	+14%	217.44	+9%
GZIP	68.38	250.52	×2.66	94.15	+38%	81.15	+19%
SQLITE3	1.32	626.61	×473.71	2.25	+70%	2.07	+56%
TCC							
sshd	2.38	29.04	×11.2	2.87	+21%	3.01	+26%
sqlite3	0.14	7.85	×56	0.28	×2	0.42	×3

TABLE 3.2 – Temps d'exécution moyen des programmes du premier corpus de programmes (utilisation intensive du CPU) ainsi que l'impact relatif de DAMAS sur ce temps d'exécution.

à 100% sont calculés grâce à la formule suivante :

$$surcout_{execution} = \frac{temps_{execution}}{temps_{reference}} - 1$$

Premier corpus

Comme attendu, DAMAS impose un surcoût en temps d'exécution important aux programmes du premier corpus, notamment dans la configuration *non-triée*, qui utilise la représentation linéaire naïve des tables de délégation. Cette représentation implique une multiplication du temps d'exécution de ces programmes comme pour TCC qui a été ralenti par un facteur de ×11 ou pour SQLITE3 qui est le cas le plus extrême dans cette évaluation avec un ralentissement de près de ×473.

Afin de comprendre un peu les raisons du ralentissement du processus cible, nous avons ajouté des compteurs au sein de nos tables de délégation. Un compteur est associé à chaque cas de la table et compte le nombre de fois que le flot de contrôle est redirigé vers celui-ci. Un compteur principal est associé à la table elle-même et est incrémenté à chaque fois que la table est visitée. Une analyse de ces compteurs nous permet de voir que la table des retours de fonction est de loin la table la plus utilisée, ce qui est attendu puisque tous les retours de fonctions sont des branchements indirects et donc sont redirigés vers la table correspondante, contrairement aux appels de fonctions qui sont majoritairement des branchements directs. Il apparait par ailleurs que SQLITE3 a une table des retours de fonction particulièrement grande avec près de 30 000 entrées et que beaucoup de compteurs situés très loin dans la table (entre les entrées 8140 et 11 183)

ont enregistré un très grand nombre d'utilisations (399 828 utilisations pour la majorité de ces compteurs avec un maximum à 2 800 519 pour l'entrée 10 555). En plus de ces compteurs loin dans la table, de nombreux autres compteurs situés plus près du début de la table ont enregistré un nombre d'utilisations similaire. Par conséquent, en utilisant DAMAS dans sa configuration non-optimisée, le processus a perdu un temps considérable à atteindre ces entrées situées loin dans la table des retours, ce qui explique en partie le surcoût en temps d'exécution de près de $\times 473$.

Ce ralentissement est amoindri dans les configurations utilisant une forme d'optimisation de DAMAS. Typiquement, GZIP et BZIP2 pour lesquels le surcoût dans la configuration *non-triée* multiplie le temps d'exécution par respectivement $\times 2.66$ et $\times 3.7$ bénéficient grandement de ces optimisations et le surcoût descend à seulement 19% et 9% pour la configuration *triée*.

Le surcoût en temps d'exécution très grand observé lors de la compilation de SQLITE3 par TCC, soit 100% dans la configuration *arbre* et 200% dans la configuration *triée* peut être expliqué par le très court temps d'exécution du programme. Il est possible que l'impact de DAMAS sur les performances ne soit pas linéaire par rapport au temps d'exécution mais qu'un surcoût minimal incompressible apparaisse ici. Par exemple, les retours de fonctions ont pu devenir une partie importante du temps d'exécution de TCC sur une exécution si courte. Il est raisonnable de penser que sur la compilation d'un programme plus gros, TCC pourrait passer plus de temps à boucler à l'intérieur d'une même fonction, ce qui implique l'exécution de plus de code séquentiel, de branchements directs et de branchements indirects que DAMAS aurait redirigés vers une table de délégation autrement plus courte (et donc plus rapide à traverser) que la table de délégation des retours de fonctions.

Nous avons par ailleurs mesuré l'impact de DAMAS sur la compilation de SSHD par TCC puisque ce programme a une base de code bien plus large que SQLITE3. Comme nous nous y attendions, le surcoût en temps d'exécution est bien plus faible, passant de 100% à 21% pour la configuration en arbre, ce qui nous conforte dans notre idée. Toutefois, il est difficile de trouver de larges projets pouvant être compilés avec une seule ligne de commande ressemblant au code suivante :

```
1 tcc -o executable *.c -ldependance
```

En effet, la compilation modulaire ne nous intéresse pas dans le cadre de nos scénarios de test. Nous ne voulons qu'une seule et longue exécution de TCC. Par ailleurs, ce com-

Programme	Référence	Damas (non-triée)		Damas (arbre)		Damas (triée)	
	temps (s)	temps (s)	surcoût	temps (s)	surcoût	temps (s)	surcoût
NGINX	501.69	501.89	+0%	500.68	+0%	499.74	+0%
NGINX-STATIC	495.31	496.74	+0%	496.10	+0%	496.91	+0%
MOSQUITTO	4.50	19.89	×3.42	4.74	+5.33%	4.70	+4.44%

TABLE 3.3 – Temps d'exécution moyen des programmes du second corpus de programmes (applications représentatives du cas d'utilisation de DAMAS) ainsi que l'impact relatif de DAMAS sur ce temps d'exécution.

pilateur est reconnu pour ses temps de compilation très courts, rendant la tâche encore plus compliquée.

Toutefois, contrairement aux autres programmes du corpus, TCC offre de meilleures performances en utilisant des tables avec une représentation en arbre plutôt qu'une table linéaire triée. Cela s'explique par l'utilisation de la table des retours de fonction, la seule table utilisée par le programme, qui n'est pas complètement biaisé en faveur d'un nombre limité d'entrées. En effet il apparait que sur les 25 357 949 cas d'utilisations de cette table, seulement 12 494 411 cas d'utilisations ciblent les 13 premières entrées de la table. Cette table comptant 7304 entrées, la représentation en arbre permet d'accéder à n'importe quelle entrée en au maximum $\lceil \log_2(7304) \rceil$ étapes, soit 13 étapes. Par conséquent, plus de la moitié des cas d'utilisations de la table sous forme linéaire triée nécessitent plus de 13 étapes, rendant la représentation en arbre plus intéressante.

Deuxième corpus

Contrairement aux programmes ayant une utilisation intensive du CPU, les serveurs souffrent beaucoup moins de l'impact de DAMAS sur leurs performances. Puisque les serveurs sont supposés avoir une utilisation intensive des entrées et sorties (notamment le réseau), nous nous attendions à un surcoût en temps d'exécution plus faible que pour les programmes du premier corpus qui comportent un plus grand nombre de branchements.

Néanmoins, le surcoût en temps d'exécution pratiquement inexistant de NGINX et NGINX-STATIC a attiré notre attention. Selon les logs de APACHEBENCH, utilisé pour la mesure du temps d'exécution, ainsi que des vérifications manuelles avec un navigateur web, le serveur fonctionne comme prévu et permet d'accéder à des pages web. De plus, DAMAS ne nous a pas alerté de la moindre erreur ni du moindre branchement indirect

qu'il n'a pas pu traduire[7]. De plus, nous avons inspecté le flot de contrôle de l'exécution de ces programmes afin de nous assurer que jamais il ne retourne dans son flot de contrôle d'origine et échappe à nos protections.

Finalement, nos résultats ne montrent pas de vraie différence entre NGINX et NGINX-STATIC en terme de temps d'exécution. En conclusion, il semble que l'impact des opérations d'entrée et sortie soit assez important pour éclipser totalement l'impact de DAMAS peu importe la différence de couverture du code par notre outil rendue possible par la compilation en statique de la grande majorité des bibliothèques utilisées par NGINX.

Troisième corpus

Notre troisième corpus correspond à un sous-ensemble de la suite de benchmark SPEC CPU 2006. Il s'agit des programmes C dont le temps d'exécution n'a pas explosé en présence de DAMAS[8]. La table 3.4 résume les temps d'exécutions moyens obtenus. Nous avons précisé les entrées utilisées, pour les programmes pour lesquels il existait plusieurs entrées ou jeux de données possibles fournis par SPEC CPU 2006. C'est par exemple le cas de BZIP2 et ses nombreux fichiers proposés comme entrée, ou H264REF que nous n'avons testé qu'avec le fichier `foreman_ref_encoder_baseline.cfg` en raison de son long temps d'exécution.

Tout comme pour le premier corpus de programmes, nous observons une nette différence de surcoût en temps d'exécution entre la version non-optimisée de DAMAS et les versions optimisées. De la même manière, nous observons des différences de performances entre une représentation des tables de délégation en arbre ou en toboggan trié. En fonction des programmes analysés, l'une de ces deux versions offre de meilleures performances que l'autre, mais aucune des deux ne s'impose universellement sur l'autre. Sur la majorité des programmes du corpus, la différence est minime et de l'ordre de quelques pourcents de temps d'exécution supplémentaire par rapport au temps d'exécution de référence. Néanmoins, la différence est plus marquée pour GOBMK et SJENG, surtout pour le premier. Pour GOBMK, il apparaît très clairement que la représentation en arbre offre de bien meilleures

7. Afin d'éviter à tout prix de faire planter le processus cible, DAMAS est très prudent dans sa traduction des branchements indirects. Si à cause d'un bogue dans DAMAS ou une imprécision dans le CFG, il n'est pas possible de traduire un branchement correctement, DAMAS ne le traduit pas et écrit un message d'erreur. Ce comportement permet au processus cible d'échapper aux protections de DAMAS et retourner exécuter son code d'origine au lieu d'être arrêté à cause de cette erreur.

8. GCC et OMNETPP ont été écartés à cause de leur temps d'exécution trop long pour le déroulement de nos tests. GCC prenait plusieurs jours à terminer pour certains fichiers en entrée tandis que nous n'avons jamais atteint la terminaison de OMNETPP sur un premier test.

Programme	Référence	Damas (non-triée)		Damas (arbre)		Damas (triée)	
	temps (s)	temps (s)	surcoût	temps (s)	surcoût	temps (s)	surcoût
ASTAR							
BigLakes2048.cfg	270.20	1032.26	×3.82	342.00	+26.57%	318.05	+17.71%
rivers.cfg	148.88	519.42	×3.49	179.36	+20.47%	169.22	+13.67%
BZIP2							
input.source 280	91.70	222.92	×2.43	128.50	+40.14%	124.14	+35.38%
chicken.jpg 30	120.36	297.74	×2.47	168.35	+39.87%	161.77	+34.40%
liberty.jpg 30	33.41	69.63	×2.08	46.97	+40.59%	47.13	+41.07%
input.program 280	51.14	105.90	×2.07	87.51	+71.12%	87.55	+71.21%
text.html 280	123.61	300.49	×2.43	169.66	+37.25%	167.32	+35.35%
input.combined 200	132.10	327.80	×2.48	229.09	+73.43%	220.76	+67.12%
GOBMK							
13x13.tst	100.77	8222.43	×81.60	214.48	×2.13	299.15	×2.97
nngs.tst	73.48	5957.33	×81.07	154.37	×2.10	217.71	×2.96
score2.tst	188.06	15460.36	×82.21	398.88	×2.12	564.69	×3
trevorc.tst	89.42	6675.32	×74.65	172.02	+92.38%	260.08	×2.91
trevord.tst	73.35	5964.37	×81.31	154.14	×2.10	223.37	×3.05
HMMER							
nph3.hmm	1.64	13.89	×8.46	1.84	+11.96%	1.79	+9.26%
retro.hmm	4.66	17.09	×3.67	4.88	+4.73%	4.85	+4.08%
H264REF							
foreman ref encoder baseline	430.33	190323.01	×442.27	1055.56	×2.45	1047.13	×2.43
LIBQUANTUM	346.06	613.06	+77.16%	582.75	+68.40%	576.70	+66.65%
MCF	242.33	308.94	+27.51%	295.36	+21.91%	294.23	+21.45%
SJENG	545.00	13529.69	×24.83	1373.06	×2.52	1260.99	×2.31
SPECRAND	0.10	0.13	+35.10%	0.13	+35.47%	0.13	+32.74%

TABLE 3.4 – Temps d'exécution moyen des programmes du troisième corpus de programmes (SPEC CPU 2006) ainsi que l'impact relatif de DAMAS sur ce temps d'exécution.

performances. Nous pouvons en conclure que l'utilisation des tables de ce programme n'est pas complètement biaisée en faveur de quelques entrées de celles-ci. À l'inverse, SJENG montre de meilleures performances si les tables de délégation sont injectées sous la forme de toboggans pré-triées de manière à mettre les entrées les plus utilisées en premier.

Les performances observées de BZIP2 ne correspondent pas à celles observées dans le premier corpus. Il y a deux principales raisons à cela. Tout d'abord, ces deux programmes ne sont pas les mêmes. Le BZIP2 du premier corpus est une copie conforme du logiciel de compression et décompression de données tandis que celui de SPEC CPU 2006 est un programme de benchmark exécutant un certain nombre de traitements dans l'objectif de faire exécuter du code par la machine. De plus, les fichiers utilisés en entrée ne sont pas les mêmes. Lors de nos tests sur le premier corpus, nous avons fait compresser un fichier ISO de plusieurs centaines de mégaoctets à BZIP2 alors que les fichiers fournis par SPEC CPU 2006 sont relativement petits. Par conséquent nous avons fait compresser ces fichiers par le BZIP2 du premier corpus afin de comparer les résultats. La majorité des cas de test prennent une fraction de seconde à s'exécuter, les surcoûts sont donc très difficiles à évaluer sérieusement [9]. Les résultat obtenus pour `input.source` et `input.combined` montrent des surcoûts en temps d'exécution similaires à ceux présentés dans l'étude du troisième corpus (67.12% de surcoût en temps d'exécution pour `input.combined` avec la version *triée* de DAMAS). Néanmoins les temps d'exécutions sont faibles (quelques secondes), aussi les surcoûts mesurés étaient très variables (du simple au double), ne permettant pas de juger précisément. Finalement, il serait plus intéressant de diversifier la nature des fichiers d'entrée tout en utilisant des fichiers de plus grandes tailles. Néanmoins nous n'avons pas eu le temps de mener une telle expérimentation.

9. Le surcoût était pratiquement nul, avec parfois une amélioration des performances avec DAMAS en version non-optimisée.

Chapitre 4

Travaux futurs

Après plusieurs années de développement, notre approche ainsi que son implémentation, Damas, nous ont permis de déployer un premier niveau de protection sur des exécutables, durant leur exécution. Nous avons toutefois identifié un certain nombre de limitations aussi bien techniques que conceptuelles.

Dans ce chapitre, nous discutons des limitations que rencontre encore notre outil et donnons des pistes d'améliorations possibles, aussi bien du point de vue des performances offertes par Damas que des aspects conceptuels qui fondent notre approche.

4.1 Tirer un meilleur profit de Rust

Rust est un langage qui comporte des fonctionnalités intéressantes pour la sécurité informatique. Ce langage offre une approche de l'intégrité de la mémoire intéressante ne nécessitant pas de ramasse-miettes. Ce mécanisme consiste essentiellement à donner à toute référence une durée de vie et une notion de *propriété*. Seule une référence possède une valeur et toute autre référence ne fait que l'emprunter, garantissant l'intégrité des données manipulées par plusieurs fils d'exécution. En effet, il ne peut exister qu'un seul propriétaire d'une variable à la fois et toute référence à cette variable ne fait que *l'emprunter*. Par défaut, Rust utilise une sémantique de mouvement et ne copie pas les valeurs lors d'une affectation. Le code suivant illustre ce principe :

```
1 // Ici, x est propriétaire de la variable valant 5.
2 let x = 5;
3 // x n'est plus valide, la variable est bougée vers y.
4 let y = x;
5 // y existe toujours mais est empruntée par z.
6 let z = &y;
```

Les références vers des variables peuvent être muables ou immuables, c'est-à-dire que

la référence peut permettre ou non de modifier la variable référencée. Afin de s'assurer de l'intégrité des données dans le contexte d'applications multithreadées, il n'est pas possible d'avoir plus d'une référence muable à la fois et il n'est pas possible d'emprunter une variable si elle a déjà été empruntée de manière muable :

```
1  // x est muable, on peut modifier la variable.
2  let mut x = 5;
3  x = 6;
4
5  // On peut emprunter x de manière muable.
6  let y = &mut x;
7  *y = 5;
8
9  // Ce code est invalide puisque x a été empruntée par y.
10 let z = &x;
11 *y = 1;
```

Ces propriétés servent à garantir l'intégrité de la mémoire en forçant le développeur à écrire du code conforme à ce qu'attend le *borrow-checker*. Ce mécanisme du compilateur analyse le code afin de vérifier que les emprunts de variables sont tous légaux.

Toutefois, lorsque nous avons commencé à développer Sorry, nous n'avons pas effectué les bons choix architecturaux nous permettant de tirer profit efficacement de ce système de durées de vie et de propriété. Par exemple, notre type de données `TargetProcess` représente un processus cible. Nous pouvons utiliser les différentes fonctionnalités de Rust pour donner du sens à ce type de données. Par exemple, il n'existe qu'un seul exemplaire du processus cible, par définition. Par conséquent, un `TargetProcess` ne doit pas pouvoir être copié. Il n'implémente donc ni le trait `Clone` ni le trait `Copy` qui permettent de copier un objet, y compris ses attributs privés. Actuellement, un `TargetProcess` contient des informations sur l'état du processus cible mais délègue à un `TargetController` la charge d'effectuer les manipulations sur le processus. Cette façon de procéder est influencée par la programmation orientée objet. Toutefois, une autre approche bien plus intuitive, efficace et sûre peut être imaginée en Rust.

L'appel système `ptrace` utilisé par le `TargetController` pour contrôler l'exécution du processus cible nécessite que ce dernier soit en pause pour effectuer certains traitements. Par exemple, `ptrace` ne peut pas lire ou modifier les registres du processus cible alors

que celui-ci est en cours d'exécution. Il ne peut pas non plus arrêter un processus déjà en pause. Notre architecture logicielle actuelle nous oblige à nous assurer explicitement que le processus cible est dans de bonnes conditions pour effectuer nos traitements. Néanmoins, RUST nous permet d'intégrer ces vérifications lors du typage et du *borrow-checking*.

Le processus cible pourrait toujours être représenté par un `TargetProcess`. Cet objet serait notamment muni d'une méthode `wait` dont le travail serait d'attendre que le processus cible soit arrêté (ce serait une encapsulation de l'appel système `waitpid`) :

```
1 impl TargetProcess {
2   pub fn wait(&mut self) -> Result<TargetController, Error> {
3     let status = waitpid(self.pid, None)?;
4     Ok(TargetController {
5       process: self,
6       reason: status
7     })
8   }
9 }
```

Cette méthode aurait deux particularités. La première serait de nécessiter une référence muable au `TargetProcess` pour être appelée (il n'est pas possible d'écrire `process.wait()` si `process` est une référence constante). La seconde serait que cette méthode renvoie un `TargetController` dont voici une définition :

```
1 pub struct TargetController<'a> {
2   process: &'a TargetProcess,
3   reason: WaitStatus
4 }
```

L'attribut `reason` de `TargetController` correspond à la valeur de retour de `waitpid` et donne la raison pour laquelle le processus a été arrêté. Il permet d'indiquer par exemple si le processus cible est en pause (arrêté via un `SIGSTOP` ou un `SIGTRAP` par exemple) ou s'il a terminé son exécution. L'attribut `process` est ici beaucoup plus intéressant. Il s'agit d'une référence constante vers un `TargetProcess`. L'intérêt de cette référence est de faire en sorte que tant que le contrôleur existe (c'est-à-dire qu'il existe une référence vivante à cette variable), le `TargetProcess` correspondant est emprunté via une référence constante. Or, le modèle mémoire de RUST stipule qu'il n'est pas possible d'utiliser une

référence muable à un objet tant qu'il est emprunté par d'autres références immuables [1]. Par conséquent, tant qu'un contrôleur existe, il est strictement impossible d'appeler la méthode `wait` du `TargetProcess` correspondant. Il n'est donc pas possible d'arrêter une deuxième fois un processus déjà arrêté. Par ailleurs, `TargetController` pourrait être muni d'une méthode `resume` avec la particularité suivante : elle ne prend pas le contrôleur en paramètre par référence comme `wait`, mais par valeur.

```
1 impl<'a> TargetController<'a> {
2   pub fn resume(self) -> Result<(), Error> {
3     let pid = self.process.pid();
4     ptrace_cont(pid, None)
5   }
6 }
```

Puisque Rust utilise une sémantique de mouvement au lieu de copies, passer un paramètre par valeur implique que sa durée de vie est réduite à l'appel de la fonction. Autrement dit, lorsqu'un contrôleur appelle sa méthode `resume` pour relancer l'exécution du processus cible, il en meurt, rendant ainsi sa liberté au `TargetProcess` et lui permettant d'appeler à nouveau sa méthode `wait` et ainsi de suite.

Grâce à ce tandem d'objets, il est possible alors de séparer les traitements à effectuer au processus cible en fonction de son état. Par exemple, il est nécessaire que le processus soit arrêté pour modifier les valeurs des registres grâce à `ptrace`, ces traitements seront donc des méthodes de `TargetController`.

4.2 Meilleure couverture des programmes existants

Damas est un prototype développé dans le cadre de nos recherches qui, contrairement à un outil de qualité industrielle, ne couvre pas l'ensemble des mécanismes utilisés dans les applications C/C++. Nous discutons par la suite de certaines limitations techniques qui nécessiteraient un effort d'ingénierie supplémentaire afin d'augmenter la couverture de notre outil.

Tout d'abord, notre outil a une utilisation intensive de notre bibliothèque de DBM, Sorry. Or, cette bibliothèque utilise régulièrement les interfaces C des appels systèmes qu'elle manipule. Par conséquent, Damas ne peut actuellement être déployé que sur des

1. Il n'est pas possible d'avoir deux références muables d'un même objet simultanément non plus.

programmes liés dynamiquement à la bibliothèque C [2]. Par exemple, des programmes écrits en Go ou en Pascal ne peuvent actuellement pas être protégés par Damas.

Toutefois, cette limitation peut aisément être contournée, moyennant un effort d'ingénierie supplémentaire. Il est possible de développer un module pour Sorry qui manipulerait des appels systèmes dans le processus cible via l'instruction `syscall` et non plus via leur interface C.

Par ailleurs, la gestion des exceptions en C++ ne se fait pas correctement si Damas est déployé sur le processus incriminé. Nous avons développé un module permettant de modifier les informations contenues dans les sections `.eh_frame` et `.eh_frame_hdr` chargées dans la mémoire du processus, afin de nous assurer que les informations manipulées par le gestionnaire d'exception correspondent au code exécuté. Toutefois, ce module ne fonctionne pas correctement à l'heure actuelle. Il n'est donc pas possible, pour le moment, d'utiliser Damas sur les programmes utilisant ce genre de mécanismes. Une suite au développement de notre outil devrait s'intéresser à ce problème.

Les programmes faisant usage de compilation JIT ne peuvent également pas être protégés par Damas. En effet, le code produit par le JIT n'existe pas dans le fichier binaire du programme, par définition. De ce fait, Damas n'est pas capable de l'intégrer à sa représentation interne du programme cible et donc aucune table de délégation n'est en mesure de contenir la moindre référence à du code JIT menant donc à des faux positifs. Malheureusement, analyser le code après chaque génération de code JIT pourrait donner à l'attaquant une possibilité de tromper Damas et lui faire accepter comme valide du code qu'il aura construit via le compilateur JIT. Bien que des réponses à ces interrogations puissent être trouvées, elles n'ont pas fait partie de nos recherches et pourraient faire l'objet de travaux ultérieurs. Des travaux tels que Ansel et al. 2011 ; Niu et Tan 2014b pourraient servir de base de réflexion.

Enfin, de nombreux programmes, parmi lesquels les interpréteurs lua, python et mruby, initialement dans notre corpus de programmes de test, respectant les quelques contraintes énoncés dans cette section continuent de planter lorsque Damas est déployé. Dans la majorité des cas, il s'agit d'une erreur de segmentation. Bien que les causes exactes de ces crashs nous soient encore inconnues, nous espérons pouvoir régler ces problème avec un débogage plus complet de Damas afin de faire de notre prototype de recherche un outil réellement utilisable.

2. Ceci est la raison pour laquelle notre programme de test nginx-static n'est pas compilé totalement en statique mais lie la bibliothèque C dynamiquement.

4.3 Optimisations du code actuel

Damas est un outil complexe et dont le code est un réseau de modules interconnectés qui ont connu un développement organique. En effet, bien que l'architecture globale de notre approche eu été connue dès le début du développement, les difficultés techniques et les cas aux limites décrits au chapitre 2 ne nous sont apparus qu'au fur et à mesure. Une des premières tâches qui pourraient être accomplies pour l'améliorer serait d'en simplifier le fonctionnement. Damas effectue de nombreuses tâches qui pourraient être considérées aujourd'hui comme superflues ou inutilement compliquées.

Par exemple, dans la sous-sous-section 2.3.3, nous évoquions la difficulté de calculer précisément la taille du prologue des tables de délégation des appels de fonctions. Nous avons pris soin durant le développement de Damas de minimiser l'impact de notre approche sur la consommation mémoire du processus cible. Malheureusement ce parti pris nous a amené à écrire une grande quantité de code compliqué et dont l'impact sur les performances de Damas restent encore à évaluer. Il apparaît néanmoins que l'appel système `mmap` que nous utilisons pour allouer la mémoire de nos `CodeCache` peut être utilisé de manière bien plus gourmande sans pénalité. En effet, `mmap` prépare des emplacements mémoire dans l'espace d'adressage du processus cible de manière à pouvoir les utiliser par la suite. Toutefois, ces emplacements mémoire ne sont réellement alloués que s'ils sont utilisés. Par conséquent il serait possible de réclamer plusieurs gigaoctets de mémoire pour nous assurer que Damas soit en mesure d'injecter tout ce dont il a besoin tout en ayant une consommation mémoire par le processus cible similaire à celle que nous avons maintenant.

Par ailleurs, Damas utilise actuellement beaucoup de `CodeCache` différents pour injecter du code. Typiquement, chaque table de délégation a son propre cache et toute la section `.secure_text` a son propre cache également. Chaque cache est allouée via un appel à `mmap`. Cela signifie que tous les caches sont placés sur des pages mémoires différentes et ce même si leur taille ne le justifie pas. Il serait intéressant de ne plus gérer ces caches indépendamment et plutôt d'avoir un grand cache unique au sein duquel les différents objets injectés seraient stockés de manière plus compacte. Ce faisant, nous serions en mesure de consommer beaucoup moins de mémoire dans le processus cible. De plus, les optimisations décrites à la section 4.4 permettront de réduire encore plus la taille des différentes tables de délégations, rendant encore plus flagrant le gaspillage de mémoire lié à la gestion individuelle des caches. Ainsi nous pourrions les compacter davantage et donc

gagner encore plus de place.

Il est important de mentionner aussi que la création d'un nouveau `CodeCache` impose aussi une pénalité sur les performances. En effet, un `CodeCache` est créé en instrumentant le processus cible pour qu'il appelle successivement les appels systèmes `mmap` et `mprotect`, le tout en manipulant les valeurs des registres ainsi qu'en sauvegardant l'état de la mémoire à l'endroit où ces appels sont injectés. Cette sauvegarde permet de restaurer l'état du processus (à l'exception de la mémoire allouée) afin qu'il continue son exécution. Le déroulement de l'allocation d'un `CodeCache` est illustré par la figure 4.1. Puisque ces manipulations se déroulent dans le processus cible, il ne s'agit pas de simples appels de fonctions, chaque manipulation est effectuée depuis Damas via des appels à `ptrace`. Par conséquent, l'allocation d'un `CodeCache` est très coûteuse en temps d'exécution. Bien que l'allocation d'une poignée de caches impacte très peu les performances, un programme avec une grande quantité de fonctions peut prendre énormément de temps à être rebasé [3]

Réduire au maximum le nombre de caches afin de regrouper les objets à injecter dans le processus cible permettrait donc de réduire drastiquement le temps de préparation de Damas. Bien que ce temps de préparation n'impacte en rien le reste de l'exécution du processus cible, il peut être un critère déterminant dans la capacité de notre approche à être adoptée comme solution de protection. À défaut de pouvoir réduire le nombre de caches de code, il serait envisageable de ne plus procéder à leur allocation manuellement, comme illustré par la figure 4.1, mais d'injecter dès le départ dans le processus cible une fonction toute entière avec la même sémantique, dont le code pourrait correspondre à ceci :

```
1 size_t allocate_codecache(size_t size) {
2   size_t address = mmap(0, size, ...);
3   mprotect(address, size, PROT_READ | PROT_EXEC);
4
5   return address;
6 }
```

La création d'un cache de code se limiterait à l'appel de cette fonction avec les bons paramètres, ce qui réduirait l'impact des appels à `ptrace` nécessaires à l'instrumentalisation du processus cible au strict minimum.

3. Durant le développement de Damas, nous avons essayé de le tester sur des programmes écrits en Rust (qui compile énormément de dépendances statiquement par défaut). Il fallait plus d'une heure pour rebaser le code de ces petits programmes.

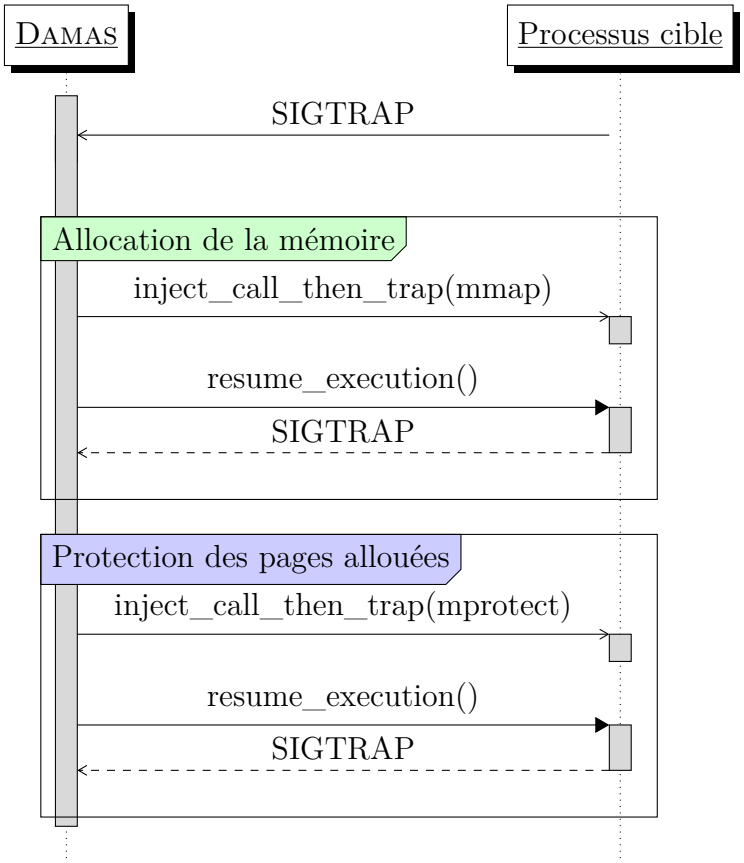

Figure 4.1 – Allocation d'un cache de code

Afin d'optimiser encore les performances offertes par Damas, il serait intéressant de changer la façon dont le code est injecté en mémoire. En effet pour l'instant le code du programme cible entier est rebasé en mémoire en une seule fois dans le but d'ajouter de l'espace entre les blocs de base. Puis, une fois cela fait, les différentes transformations du code ont lieu pour ensuite injecter les tables de délégations. Cette façon de procéder à toutefois un inconvénient majeur : les tables sont systématiquement éloignées du reste du code. Typiquement, les tables de délégation des sauts intraprocéduraux sont très éloignées des fonctions à laquelle elles appartiennent. L'utilisation du cache est donc sous-optimal et possiblement source de ralentissements. Par conséquent, injecter ces tables à proximité de leur fonction permettrait d'améliorer l'utilisation du cache et donc probablement d'améliorer les performances du processus cible.

4.4 Réduction de la taille des classes d'équivalence

Dans le chapitre 2, nous avons introduit le concept de table de délégation. Ces tables sont par essence une forme d'optimisation des tobbogans introduits par Arthur et al. 2015. En effet, l'intérêt des tables de délégation est de permettre à toutes les instructions d'une même classe d'équivalence d'utiliser le même tobbogan. Néanmoins, cette structure de contrôle est lourde et souffre de plusieurs défauts.

Un de ces défauts est la taille de ces tables. Les tobbogans de CDI sont relativement petits puisque chaque tobbogan correspond à une unique instruction de transfert de contrôle indirecte et que les cibles potentielles de cette instruction sont en principe peu nombreuses. À l'inverse, les cibles potentielles des instructions d'une même classe d'équivalence dans Damas sont nombreuses. La principale raison est la façon dont ces classes d'équivalence sont construites. Comme nous l'avons expliqué dans la sous-section 2.3.2, nous nous basons essentiellement sur trois grandes hypothèses pour construire nos classes d'équivalence. Les deux premières hypothèses ne permettent pas de définir de classes d'équivalence précises. En effet, la première hypothèse place tous les appels de fonctions dans une même classe et tous les retours de fonctions dans une autre.

Il serait alors intéressant de réduire au maximum la taille de ces classes d'équivalence. Cela permettrait de s'assurer d'une traversée plus rapide des tables de délégation avant de pouvoir traiter les cas particuliers comme les appels vers l'étranger. De plus, cela contribuerait à renforcer le niveau de sécurité apporté par le mécanisme de protection puisque réduire la taille des classes d'équivalence revient à affiner le CFG qu'elles définissent. Comme expliqué dans la section 3.1, affiner davantage notre CFG ne permettra pas de protéger les processus cibles d'attaques telles que *Control-Flow Bending*. Toutefois, cela rendrait de telles attaques bien plus difficiles à mettre en place, puisque la recherche de gadgets viables deviendrait plus compliquée elle aussi.

Dans un premier temps, il est possible de réduire grandement la taille des classes d'équivalence de sauts intraprocéduraux. En effet, afin de nous assurer du bon fonctionnement des processus cibles et éviter au maximum les faux positifs, nous avons été très prudents concernant les sauts indirects. Par conséquent nous avons considéré que toute adresse à l'intérieur d'un bloc de base d'une même fonction était une adresse cible valide. Néanmoins de nombreux travaux se sont intéressés à la reconstruction de graphes de flot de contrôle depuis le binaire (Bauman, Lin et Hamlen 2018 ; Shoshitaishvili et al. 2016) et bien que ces approches ne permettent pas de reconstruire des CFGs parfaits,

leurs résultats pourraient être utile localement. Nous avons par exemple utilisé le framework ANGR (SHOSHITAISHVILI et al. 2016) pour reconstruire les CFGs de programmes de nos corpus de test. Il apparaît que ANGR est parfaitement capable de retrouver les cibles de sauts indirects correspondant à un `switch` dans le code source. Partant de cette observation, il serait possible de réduire le tobbogan correspondant à ces instructions aux cibles identifiées. Par ailleurs, puisqu'une seule instruction peut servir de répartiteur pour un `switch`, il est inutile de construire toute une classe d'équivalence dont on sait qu'elle ne comportera que cette instruction. Par conséquent, il serait préférable dans ces cas de construire un simple tobbogan, ce qui pourrait en plus améliorer l'utilisation du cache par le processus cible.

De plus, comme expliqué dans la sous-section 2.3.3, certains sauts indirects se révèlent en réalité être des appels de fonctions [4]. Bien qu'il soit particulièrement difficile d'identifier ce genre de branchements depuis un fichier binaire dénué de symboles et d'informations de débogage, il est au moins possible de les identifier à l'exécution. En effet, dans un programme écrit via un langage de haut-niveau puis compilé, un saut inconditionnel ne peut pas représenter à la fois un branchement intraprocédural et un appel de fonction. Ainsi, à l'exécution, si un saut indirect se trouve être un appel de fonction, DAMAS peut facilement obtenir l'information. En effet, les appels récursifs terminaux sont traités par le code d'erreur à la fin des tables de délégations des sauts inconditionnels (qui les redirige actuellement vers la table des appels de fonctions). Ce même code d'erreur pourrait aussi demander à DAMAS de traduire l'instruction de saut vers la table des sauts de la fonction, de manière à ce qu'elle cible désormais la table des appels de fonction directement. Ainsi, il ne serait nécessaire de traverser l'intégralité de la table des sauts de la fonction puis la table des appels de fonctions qu'une seule fois. Les fois suivantes, seule la table des appels de fonctions serait traversée.

De la même manière, un mécanisme pourrait faire en sorte qu'une fois qu'un saut a été confirmé comme saut intraprocédural [5], le code d'erreur de la table ne peut plus traduire le saut en appel de fonction via le mécanisme présenté au paragraphe précédent. Ce mécanisme améliorerait alors d'autant plus la sécurité apportée par DAMAS.

4. Ce sont les résultats d'une optimisation des appels récursifs terminaux.
5. Un saut est confirmé comme intraprocédural s'il cible une instruction dans la même fonction. Selon l'hypothèse qu'un saut ne peut être à la fois un saut intraprocédural et un appel récursif terminal, il suffit d'une exécution du saut pour en connaître la nature.

Conclusion

Durant cette thèse, nous avons sommes intéressés aux attaques par corruption de mémoire et avons développé une approche de protection du flot de contrôle d'un processus sans nécessiter de redémarrage de ce dernier. Notre approche prend la forme d'un framework de modification dynamique de binaire capable d'injecter des protections durant l'exécution du processus cible. Nous avons développé un outil utilisant ce framework dans l'objectif de mettre en place une protection dérivée de CDI (Arthur et al. 2015). Le développement de cet outil nous a permis d'explorer les possibilités offertes par la mise en place de protections durant l'exécution du programme cible.

Par la suite, nous résumons les contributions apportées par nos travaux ainsi que les pistes d'améliorations possibles pour de futures recherches.

Nos contributions

Nos recherches ont permis d'aboutir à des contributions dans le domaine de la compilation avec une bibliothèque de DBM ainsi que dans le domaine de la sécurité en décrivant une approche de protection de processus en cours d'exécution et un outil de protection basé sur notre approche. Dans cette section, nous revenons brièvement sur ces trois contributions et expliquons l'intérêt de celles-ci.

Sorry, notre bibliothèque de DBM

Afin de développer notre framework d'injection de protections, nous avons créé une bibliothèque de DBM généraliste. L'intérêt de cette bibliothèque est de nous libérer de l'impact sur les performances de projets plus lourds comme Pin ou DynamoRIO. Ces frameworks effectuent diverses transformations sur le code du processus cible dans le but de l'instrumenter et ces manipulations ont un coût. Notre bibliothèque, à l'inverse ne fait rien sans ordre explicite de l'utilisateur, ce qui permet de contrôler au maximum l'impact de notre solution sur les performances du processus cible.

De plus, contrairement aux autres bibliothèques de DBM dont nous avons la connais-

sance, nous n'avons pas écrit Sorry en C ou en C++ mais en Rust. Ce choix de langage apporte des garanties quant à l'intégrité de la mémoire du client Sorry et permet un développement plus simple que dans des langages de plus bas niveau. Des abstractions telles que les `Buffers` permettant de raisonner davantage sur les traitements à effectuer que sur leur bonne conduite. Tous ces avantages de Rust ont permis de simplifier grandement le débogage, comparé à nos essais avec notre ancienne bibliothèque Padrone.

En conclusion, notre bibliothèque permet l'écriture d'outils de DBM puissants et légers de manière simple tout en portant une attention particulière à l'impact de ces outils sur les performances du processus cible. L'intérêt d'une telle approche est de permettre une instrumentation légère du processus cible sans prendre totalement le contrôle de son exécution et préservant ainsi au maximum ses performances, contrairement à des outils plus intrusifs tels que Pin ou DynamoRIO.

Damas, notre framework de protection

Nous avons développé une approche dont l'objectif est de pouvoir protéger un processus contre les attaques par corruption de mémoire alors que celui-ci est déjà en cours d'exécution. Nous avons implémenté cette approche sous la forme d'un framework, Damas. Notre framework met en place les protections voulues en quelques étapes.

Tout d'abord, il prend le contrôle du processus cible via `ptrace`. Puis, le code du programme cible est désassemblé dans son intégralité, ce qui permet de récupérer des informations cruciales telles que le code lui-même, les adresses des fonctions voire un CFG. Ensuite, le code est instrumenté afin d'ajouter les vérifications nécessaires, puis il est injecté dans l'espace adressage du processus cible. Enfin, le flot de contrôle du processus cible est redirigé vers le code instrumenté et Damas commence une surveillance continue de l'exécution du processus. Cette surveillance permet de garantir la sécurité du programme et possiblement d'améliorer les performances des protections injectées.

L'avantage de cette approche par rapport à d'autres protections de type CFI est de ne pas nécessiter de modifier le programme avant de l'exécuter. En effet, d'ordinaire le fichier binaire du programme est édité dans le but d'y ajouter du code d'instrumentation. Toutefois, Damas ne modifie que l'image du processus chargé en mémoire pendant son exécution. Par ailleurs, cette technique permet de protéger un processus alors que son exécution est déjà en cours. Ainsi, il n'est pas nécessaire de l'arrêter ni de le relancer, avec toutes les conséquences que le redémarrage de ce processus pourrait avoir (accessibilité à un système critique, conséquences économiques, perte de temps, etc).

Protection dérivée de CDI à l'exécution

Nous avons adapté et porté dans Damas la protection CDI, une approche de CFI dont la particularité est de supprimer les branchements indirects du code du programme cible au lieu de les entourer de vérifications. Cette approche transforme les branchements indirects en tobbogans de pairs comparaison/branchement direct correspondant aux cibles valides du branchement indirect.

Originellement, CDI est mis en place à la compilation. Par conséquent, les tobbogans sont construits en utilisant le CFG fourni par le compilateur. Notre framework utilise le code binaire du programme cible, il a donc été nécessaire d'adapter CDI à notre cas d'usage. Puisqu'il est très difficile d'obtenir un CFG complet et minimal depuis le binaire, nous avons travaillé sur un CFG à très gros grain. À cause du grand nombre de cibles possibles définies par ce CFG pour chaque branchement indirect, les tobbogans correspondants prendraient énormément de place en mémoire. Par conséquent nous avons défini des classes d'équivalence entre les branchements indirects du programme, permettant de ne créer qu'un seul tobbogan vers lequel les branchements d'une même classe d'équivalence sont redirigés. À partir de ces classes d'équivalence nous avons défini une structure de contrôle évoluée basée sur les tobbogans de CDI que nous avons appelée tables de délégation.

Nous avons présenté des optimisations permettant de réduire grandement l'impact de notre outil sur les performances du processus cible. Ces optimisations consistent à modifier la représentation des tables de délégation de manière à rendre leur traversée plus rapide. Bien que cela permette de réduire le surcoût en temps d'exécution à moins de quatre fois le temps d'exécution normal du programme, ce n'est pas encore suffisant pour l'adoption de notre outil dans l'industrie. Dans la section suivante, nous décrivons quelques pistes d'amélioration.

Travaux futurs

Tout d'abord, notre bibliothèque, Sorry, ne tire pas pleinement profit des possibilités offertes par le langage Rust. Il serait envisageable de réécrire la bibliothèque, riches de notre expérience avec cette première version imparfaite. Il serait intéressant par ailleurs de corriger diverses limitations techniques actuelles de la bibliothèque telles que sa dépendance forte envers la bibliothèque standard du C.

Notre approche mérite elle aussi d'être approfondie. Durant nos travaux, nous avons été

très prudents vis à vis des informations que Damas utilise pour mener à bien ses missions. En effet, nous voulons garantir à l'utilisateur que non seulement le bon fonctionnement de son processus n'est pas perturbé par nos modifications mais aussi que Damas n'utilise pas des informations erronées ou malveillantes permettant à un attaquant de le mettre en défaut. Par conséquent, Damas n'analyse ni les chemins empruntés par le processus pendant l'exécution pour obtenir des informations sur le CFG du programme ni le contexte d'exécution pour rendre son CFI plus précis. Néanmoins, d'autres CFI existants en sont déjà capables (Van der Veen et al. 2015) et il serait intéressant d'explorer cette piste.

Bibliographie

Abadi, Martin, Mihai Budiu, Úlfar Erlingsson et Jay Ligatti (2005), « Control-Flow Integrity », in : *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, Alexandria, VA, USA : Association for Computing Machinery, p. 340-353, isbn : 1595932267, doi : 10.1145/1102120.1102165, url : https://doi.org/10.1145/1102120.1102165.

Andriesse, Dennis, Xi Chen, Victor Van Der Veen, Asia Slowinska et Herbert Bos (2016), « An in-depth analysis of disassembly on full-scale x86/x64 binaries », in : *25th {USENIX} Security Symposium ({USENIX} Security 16)*, p. 583-600.

Ansel, Jason, Petr Marchenko, Ulfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle et Bennet Yee (2011), « Language-independent sandboxing of just-in-time compilation and self-modifying code », in : *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, p. 355-366.

Ap, Arif Ali, Kévin Le Bon, Byron Hawkins et Erven Rohou (2018), « FITTCHOOSER : A Dynamic Feedback Based Fittest Optimization Chooser », in : *2018 International Conference on High Performance Computing Simulation (HPCS)*, p. 98-105, doi : 10.1109/HPCS.2018.00031.

Arthur, William, Ben Mehne, Reetuparna Das et Todd Austin (2015), « Getting in control of your control flow with control-data isolation », in : *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, p. 79-90, doi : 10.1109/CGO.2015.7054189.

Aycock, John (juin 2003), « A Brief History of Just-in-Time », in : *ACM Comput. Surv.* 35.*2*, p. 97-113, issn : 0360-0300, doi : 10.1145/857076.857077, url : https://doi.org/10.1145/857076.857077.

Baratloo, Arash, Navjot Singh et Timothy K Tsai (2000), « Transparent run-time defense against stack-smashing attacks. », in : *USENIX Annual Technical Conference, General Track*, p. 251-262.

Bauman, Erick, Zhiqiang Lin et Kevin W Hamlen (2018), « Superset Disassembly : Statically Rewriting x86 Binaries Without Heuristics. », in : *NDSS*.

Bletsch, Tyler, Xuxian Jiang, Vince W Freeh et Zhenkai Liang (2011), « Jump-oriented programming : a new class of code-reuse attack », in : *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, p. 30-40.

Bruening, D., T. Garnett et S. Amarasinghe (2003), « An infrastructure for adaptive dynamic optimization », in : *International Symposium on Code Generation and Optimization, 2003. CGO.* P. 265-275, doi : 10.1109/CGO.2003.1191551.

Carlini, Nicholas, Antonio Barresi, Mathias Payer, David Wagner et Thomas R. Gross (2015), « Control-Flow Bending : On the Effectiveness of Control-Flow Integrity. », in : *USENIX Security Symposium*, p. 161-176.

Carvalho, Marco, Jared DeMott, Richard Ford et David A Wheeler (2014), « Heartbleed 101 », in : *IEEE security & privacy* 12.*4*, p. 63-67.

Castro, Miguel, Manuel Costa et Tim Harris (2006), « Securing software by enforcing data-flow integrity », in : *Proceedings of the 7th symposium on Operating systems design and implementation*, p. 147-160.

Checkoway, Stephen, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham et Marcel Winandy (2010), « Return-oriented programming without returns », in : *Proceedings of the 17th ACM conference on Computer and communications security*, p. 559-572.

Chevalier-Boisvert, Maxime et Marc Feeley (2015a), *Interprocedural Type Specialization of JavaScript Programs Without Type Analysis*, arXiv : 1511.02956 [cs.PL].

— (2015b), *Simple and Effective Type Check Removal through Lazy Basic Block Versioning*, arXiv : 1411.0352 [cs.PL].

Duck, Gregory J, Xiang Gao et Abhik Roychoudhury (2020), « Binary rewriting without control flow recovery », in : *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 151-163.

Evans, Isaac, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi et Stelios Sidiroglou-Douskos (2015), « Control jujutsu : On the weaknesses of fine-grained control flow integrity », in : *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, p. 901-913, isbn : 1-4503-3832-1.

Gawlik, Robert et Thorsten Holz (août 2018), « SoK : Make JIT-Spray Great Again », in : *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD : USENIX Association, url : https://www.usenix.org/conference/woot18/presentation/gawlik.

Hallou, Nabil (déc. 2017), « Runtime optimization of binary through vectorization transformations », Theses, Université Rennes 1, url : https://tel.archives-ouvertes.fr/tel-01795489.

Hallou, Nabil, Erven Rohou et Philippe Clauss (juin 2017), « Runtime Vectorization Transformations of Binary Code », in : *International Journal of Parallel Programming* 8.*6*, p. 1536-1565, doi : 10.1007/s10766-016-0480-z, url : https://hal.inria.fr/hal-01593216.

Hawkins, Byron, Brian Demsky et Michael B. Taylor (2016), « BlackBox : Lightweight security monitoring for COTS binaries », in : *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, p. 261-272.

Hazelwood, Kim (2011), *Dynamic Binary Modification : Tools, Techniques and Applications*.

Hu, Hong, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena et Zhenkai Liang (2016), « Data-oriented programming : On the expressiveness of non-control data attacks », in : *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, p. 969-986.

Ispoglou, Kyriakos K., Bader AlBassam, Trent Jaeger et Mathias Payer (2018), « Block Oriented Programming : Automating Data-Only Attacks », in : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Toronto, Canada : Association for Computing Machinery, p. 1868-1882, isbn : 9781450356930, doi : 10.1145/3243734.3243739, url : https://doi.org/10.1145/3243734.3243739.

Kiriansky, Vladimir, Derek Bruening et Saman Amarasinghe (août 2002), « Secure Execution via Program Shepherding », in : *11th USENIX Security Symposium (USENIX Security 02)*, San Francisco, CA : USENIX Association, url : https://www.usenix.org/conference/11th-usenix-security-symposium/secure-execution-program-shepherding.

Le Bon, Camille, Erven Rohou, Frédéric Tronel et Guillaume Hiet (2021), « DAMAS : Control-Data Isolation at Runtime through Dynamic Binary Modification », in : *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, p. 86-95, doi : 10.1109/EuroSPW54576.2021.00016.

Lu, Tingting et Junfeng Wang (2019), « Data-flow bending : On the effectiveness of data-flow integrity », in : *Computers & Security* 84, p. 365-375.

Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi et Kim Hazelwood (juin 2005), « Pin : Building Customized Program Analysis Tools with Dynamic Instrumentation », in : *SIGPLAN Not.* 40.*6*, p. 190-200, issn : 0362-1340, doi : 10.1145/1064978.1065034, url : https://doi.org/10.1145/1064978.1065034.

Mashtizadeh, Ali Jose, Andrea Bittau, Dan Boneh et David Mazières (2015), « CCFI : Cryptographically enforced control flow integrity », in : *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, p. 941-951.

MITRE (2021), *2021 CWE Top 25 Most Dangerous Software Weaknesses*, url : https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (visité le 06/01/2022).

Nethercote, Nicholas et Julian Seward (2007), « Valgrind : a framework for heavyweight dynamic binary instrumentation. », in : *PLDI*, sous la dir. de Jeanne Ferrante et Kathryn S. McKinley, ACM, p. 89-100, isbn : 978-1-59593-633-2, url : http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07.

Niu, Ben et Gang Tan (2014a), « Modular control-flow integrity », in : *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 577-587.

— (2014b), « RockJIT : Securing just-in-time compilation using modular control-flow integrity », in : *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, p. 1317-1328.

— (2015), « Per-input control-flow integrity », in : *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, p. 914-926.

One, Aleph (1996), « Smashing the stack for fun and profit », in : *Phrack magazine* 7.*49*, p. 14-16.

Payer, Mathias, Antonio Barresi et Thomas R. Gross (2015), « Fine-Grained Control-Flow Integrity Through Binary Hardening », in : *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, Milan, Italy : Springer-Verlag, p. 144-164, isbn : 9783319205496, doi : 10.1007/978-3-319-20550-2_8, url : https://doi.org/10.1007/978-3-319-20550-2_8.

Payer, Mathias et Thomas R. Gross (2013), « String Oriented Programming : When ASLR is Not Enough », in : *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, Rome, Italy : Association

for Computing Machinery, ISBN : 9781450318570, DOI : 10.1145/2430553.2430555, URL : https://doi.org/10.1145/2430553.2430555.

PRANDINI, Marco et Marco RAMILLI (2012), « Return-oriented programming », in : *IEEE Security & Privacy* 10.*6*, p. 84-87.

RIOU, Emmanuel, Erven ROHOU, Philippe CLAUSS, Nabil HALLOU et Alain KETTERLIN (jan. 2014), « PADRONE : a Platform for Online Profiling, Analysis, and Optimization », in : *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, Vienne, Austria, URL : https://hal.inria.fr/hal-00917950.

SHACHAM, Hovav (2007), « 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*, p. 552-561.

SHOSHITAISHVILI, Yan, Ruoyu WANG, Christopher SALLS, Nick STEPHENS, Mario POLINO, Andrew DUTCHER, John GROSEN, Siji FENG, Christophe HAUSER, Christopher KRUEGEL et Giovanni VIGNA (2016), « SOK : (State of) The Art of War : Offensive Techniques in Binary Analysis », in : *2016 IEEE Symposium on Security and Privacy (SP)*, p. 138-157, DOI : 10.1109/SP.2016.17.

SZEKERES, László, Mathias PAYER, Tao WEI et Dawn SONG (2013), « SoK : Eternal War in Memory », in : *2013 IEEE Symposium on Security and Privacy*, p. 48-62, DOI : 10.1109/SP.2013.13.

TICE, Caroline, Tom ROEDER, Peter COLLINGBOURNE, Stephen CHECKOWAY, Úlfar ERLINGSSON, Luis LOZANO et Geoff PIKE (août 2014), « Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM », in : *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA : USENIX Association, p. 941-955, ISBN : 978-1-931971-15-7, URL : https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

VAN DER VEEN, Victor, Dennis ANDRIESSE, Enes GÖKTAŞ, Ben GRAS, Lionel SAMBUC, Asia SLOWINSKA, Herbert BOS et Cristiano GIUFFRIDA (2015), « Practical context-sensitive CFI », in : *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, p. 927-940.

ZHANG, Mingwei et R. SEKAR (2013), « Control Flow Integrity for COTS Binaries », in : *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, Washington, D.C. : USENIX Association, p. 337-352, ISBN : 9781931971034.

GLOSSAIRE

AIR *Average Indirect-target Reduction*. 24

ASLR *Address Space Layout Randomization*. 15

borrow-checker passe du compilateur RUST responsable du borrow-checking. 94

borrow-checking analyse utilisé par RUST pour vérifier que les accès à la mémoire sont conformes au modèle mémoire proposé par le langage. 95, 107

CDI *Control-Data Isolation*. 25–27, 31, 34, 40, 41, 48, 50, 56, 57, 79, 81, 101, 103, 105

CFG *Control-Flow Graph*. 14, 20–26, 34, 50, 51, 61, 79, 81, 88, 101, 102, 104–106

CFI *Control-Flow Integrity*. 3, 20, 22–25, 29–31, 33, 48, 79–82, 85, 104–106

CPU *Central Processing Unit*. 82, 83

CWE *Common Weakness Enumeration*. 5

DBM *Dynamic Binary Modification*. 3, 7, 27–31, 35, 37, 39, 96, 103, 104

DBT *Dynamic Binary Translation*. 30

DFB *Data-Flow Bending*. 14

DFG *Data-Flow Graph*. 14

DFI *Data-Flow Integrity*. 14

DOP *Data-Oriented Programming*. 13

DSL *Domain Specific Language*. 14

DSR *Data-Space Randomization*. 16

gadget dispatcher fragment de logique dans le code qui permet de chaîner des gadgets de données disjoints de manière arbitraire. 13, 14

IFCC *Indirect Function-Call Checks*. 22, 23

JIT *Just-In-Time*. 12, 16, 97

MCFI *Modular Control-Flow Integrity*. 23

Memory Safety politique de sécurité s'assurant qu'aucun pointeur ne peut être corrompu, elle s'assure que les bornes ainsi que la durée de vie de la donnée sont respectées. 16

PLT *Procedure Linkage Table*. 4, 58, 67, 69, 70, 72

ramasse-miettes mécanisme de gestion automatique de la mémoire simulant une mémoire infinie. 93

return-to-libc attaque par détournement de flot de contrôle consistant à rediriger un retour de fonction vers une fonction de la bibliothèque C. 16, 17

ROP *Return-Oriented Programming*. 17, 20, 79

sandbox mécanisme de sécurité utilisant l'isolation du processus par rapport au système pour diminuer les risques liés à l'exécution du programme. 28

shadow stack pile parallèle à la pile du programme qui est utilisée pour s'assurer de l'intégrité des données sur la pile principale. 22

SPL *SPloit Language*. 14

tobbogan structure de contrôle permettant de transformer un branchement indirect en une série de comparaisons avec des cibles valides et branchements directs. 3, 25–27, 41, 42, 48, 50, 51, 54–58, 61–67, 72, 101, 102, 105

VTV *Virtual-Table Verification*. 22, 23

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Analyse et optimisation dynamiques de programmes au format binaire pour la cybersécurité

Mot clés : isolation des données de contrôle, modification de binaire dynamique, réécriture de binaire

Résumé : Les attaques par corruption de mémoire ont été un problème majeur dans la sécurité des logiciels depuis plus de vingt ans et restent l'un des types d'attaques les plus dangereux et les plus répandus de nos jours. Parmi ces attaques, celles par détournement de flot de contrôle sont les plus populaires et les plus puissantes, permettant à l'attaquant d'exécuter du code arbitraire dans le processus cible. De nombreuses approches ont été développées pour mitiger de telles attaques et pour les empêcher de se produire.

L'une de ces approches est l'isolation des données de contrôle (CDI) qui tente de d'empêcher ces attaques en supprimant leur déclencheur dans le code, à savoir les branchemets indirects. Cette approche a été mise en œuvre sous la forme d'une passe de compilateur remplaçant tous les branchements indirects dans le programme par une table conduisant le flot de contrôle vers des sauts directs écrits en dur. L'inconvénient de cette approche est qu'elle nécessite la recompilation du programme. Dans ce manuscript, nous présentons une approche et son implémentation, DAMAS, un framework capable de déployer des protections sur un logiciel en cours d'exécution et d'utiliser les informations disponibles pour les optimiser pendant l'exécution. Nous avons implémenté une protection CDI à gros grain à l'aide de notre framework et évalué son impact sur les performances.

Title: Dynamic Binary Analysis and Optimization for Cybersecurity

Keywords: control-data isolation, dynamic binary modification, binary rewriting

Abstract: Memory corruption attacks have been a major issue in software security for over two decades and are still one of the most dangerous and widespread types of attacks nowadays. Among these attacks, control-flow hijack attacks are the most popular and powerful, enabling the attacker to execute arbitrary code inside the target process. Many approaches have been developed to mitigate such attacks and to prevent them from happening.

One of these approaches is the Control-Data Isolation (CDI) that tries to prevent such attacks by removing their trigger from the code, namely indirect branches. This approach has been implemented as a compiler pass that replaces every indirect branches in the program with a table that leads the control-flow to direct hard-written branches. The drawback of this approach is that it needs the recompilation of the program. In this manuscript we present an approach and its implementation, DAMAS, a framework capable of deploying protections on a running software and use runtime information to optimize them during the execution. We implemented a coarse-grain CDI protection using our framework and evaluated its impact on performance.