Protections des processeurs contre les cyber-attaques par vérification de l'intégrité du flot d'exécution

Michaël Timbert

▶ To cite this version:

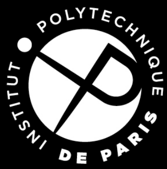
Michaël Timbert. Protections des processeurs contre les cyber-attaques par vérification de l'intégrité du flot d'exécution. Cryptographie et sécurité [cs.CR]. Institut Polytechnique de Paris, 2020. Français. NNT : 2020IPPAT028 . tel-03066435

HAL Id: tel-03066435
https://theses.hal.science/tel-03066435

Submitted on 15 Dec 2020

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.



NNT : 2020IPPAT028

Thèse de doctorat

Protections of Processors against Cyber Attacks by Control Flow Checking

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique, données, IA

Thèse présentée et soutenue à Paris 75015, le 15 Septembre 2020, par

Michaël Timbert

Composition du Jury :

David Naccache Professeur, ENS (Information Security Group)	Rapporteur, Président
Marie-Laure Potet Professeur, Grenoble INP (Ensimag)	Rapporteur
Philippe Maurine Maître de conférences/HDR, LIRMM	Examinateur
Annelie Heuser Chargée de recherche, CNRS (TAMIS)	Examinateur
Jean-Luc Danger Directeur d'études, Télécom Paris (COMELEC)	Directeur de thèse
Sylvain Guilley Professeur, Télécom Paris (COMELEC), CTO (Secure-IC)	Co-directeur de thèse
Karine Heydemann Maître de conférences/HDR, Sorbonne Université (LIP6)	Invitée
Ulrich Kühne Maître de conférences, Télécom Paris (COMELEC)	Invité

Contents

List of Figures 7

List of Tables 9

Remerciements 11

Abstract 13

1 Introduction 27

1.1 Today's processors 27

1.1.1 Common architecture of today's processors 27

1.1.2 RISC Processors 28

1.2 Control Flow Graph 28

1.3 Control Flow Hijacking 29

1.4 Control Flow Integrity 30

1.5 Introduction 33

2 State-of-the-art 37

2.1 Software weaknesses and associated attacks 37

2.1.1 Introduction 37

2.1.2 Program architecture 37

Static structures 38

Dynamic structures 39

Dynamic Loading of Libraries. 44

2.1.3 Vulnerable points 47

Spatial errors 47

Temporal errors 48

2.1.4 Vulnerability exploitation 49

Data only attack 49
Stack overflow 51
Heap overflow 54
Data and bss overflow 56
2.1.5 Payload creation method 57
Code injection 57
Code Reuse Attack 59
2.1.6 Synthesis 65
2.2 Hardware countermeasure 67
2.2.1 Existing Hardware Protections 67
Data Execution Prevention 67
Control-flow Enforcement Technology 68
Memory Protection Extension 70
Pointer Authentication 73
Trusted Execution Environment 77
TrustZone 77
Software Guard Extensions 78
Secure Memory Encryption 80
Virtual-Machine Extensions 83
Secure Virtual Machine 84
Silicon Secure Memory 85
2.2.2 Academic studies 87
Secure-Call 87
HCODE 89
Project CHERI 90
Processor SAFELite: Low fat pointer 92
HardBound 93
Compilation-Enforced Temporal Safety 94
Architectural Support for Instruction Set Randomization 95
Software and Control Flow Integrity Architecture . . . 96
Hardware-Assissted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation 98
2.2.3 CFI 98
ECCA 99
CFCSS 99
HAFGCFI 100
HCFI 100

3 Threat Model **101**
3.1 Statement of threat model 101
3.2 Mitigation of the threat 102
3.3 Offered security 102
3.3.1 Fine-grain security 102
3.3.2 Reactive vs infective strategies 103
3.4 Comparison with SOA 104

4 Functional Mechanism of Control Flow Integrity **105**
4.1 Formalism 105
4.1.1 Definitions and security property 105
4.1.2 Concept of seamless verification of security property P 107
4.1.3 Basic-block based seamless verification of property P . 109
4.2 HCODE Operating Principle 114
4.2.1 Big picture 114
4.2.2 Overview 114
4.3 Software implementation 115
4.3.1 Adding jump 116
4.3.2 The new '.HCODE' section 117
4.3.3 Hardware module 118
HCODE FSM 119
4.3.4 LEON 3 120
4.3.5 Software OS 121
HCODE verification 121
Kernel context 121
4.4 Performance estimation 122
4.4.1 Hardware performance 122
4.4.2 Software performance 123
4.5 Conclusion 123

5 CCFI: Code and Control Flow Integrity **129**
5.1 Solution 129
5.1.1 Architecture Overview 129
5.1.2 Metadata 131
5.1.3 CFI Verification 133
5.1.4 Attack Model and Security Guaranties 134
5.2 Implementation 137
5.3 Performance 138

5.4 Conclusion 140

6 Interruption and Speculative Execution 143
6.1 Solution 143
6.1.1 Hardware 144
6.1.2 Software 145
Metadata 145
Toolchain 146
6.2 Speculative execution 148
6.3 Interruptions 150
6.4 Implementation 152
6.5 Performance 152
6.6 Conclusion 153

7 Conclusion 155

8 Perspectives 157
8.1 Limitation of our approach 157
8.2 Interface with the processor 158
8.3 Metadata alignment 159
8.4 Cache 160
8.5 Basic Block Optimisation 160

Acronyms 161

9 Bibliography 163

List of Figures

1 Représentation simplifiée du graphe d'un programme 16
2 Vue simplifiée de l'architecture de base du `CCFI` 21
3 Vue simplifiée de l'architecture de la solution `CCFI` prenant en compte l'exécution spéculative 23

1.1 Example of program and associated flow graph (from [34]) . . 29

2.1 Typical data structure on the stack 40
2.2 Structure of an allocated memory *chunk* 42
2.3 Linked list of "free" memory *chunks* 43
2.4 Structure of a "free" memory *chunk* 44
2.5 Compilation and execution cycle of a program 45
2.6 2 ways of realizing *buffer overflows* 48
2.7 Overflow in the stack 52
2.8 Example of a *bin* 54
2.9 *chunk* modification in the *heap* 56
2.10 `bss` segment 57
2.11 Stack before and after a ROP attack 62
2.12 Stack during a ROP attack 63
2.13 JOP example 64
2.14 The different attack models 66
2.15 Example of *2*-level Lookup-Tree 71
2.16 Attack surface with and without the SGX protection (diagram extracted from Intel website) 79
2.17 Remote attestation (diagram from the Intel website) 80
2.18 SEV Architecture (diagram inspired by the AMD website) . . 81
2.19 Read and DWrite in SME mode (diagram from the AMD website) 82
2.20 Schematic representation of VMX 84

2.21 SVM protection representation . . . 85
2.22 SSM representation . . . 86
2.23 Schematic representation of the countermeasure SCALL . . . 88
2.24 Example of *Basic Block (BB)* without jump in the end . . . 89
2.25 HCODE hardware module . . . 91
2.26 *fat pointer* representation . . . 92
2.27 Diagram of a BIMA type counter . . . 92
2.28 Example of *low fat pointer* . . . 93
2.29 CETS schematic operation . . . 94
2.30 SOFIA operation diagram (extracted from [27]) . . . 97
2.31 *Basic Block* allowing two predecessors (diagram extracted from [27]) . . . 98

3.1 Attack vectors addressed by our CCFI technology . . . 102

4.1 Program state machine (*left*) and CFI state machine (*right*) . 108
4.2 Simple program graph representation . . . 110
4.3 Program state machine (*left*) and CFI state machine (*right*), optimized to take into account basic blocks . . . 111
4.4 Modified compilation toolchain . . . 116
4.5 Hardware implementation (simplified) . . . 118
4.6 FSM of the HCODE controller hardware module . . . 125
4.7 Gaisler board used for the LEON3 prototype . . . 126
4.8 LEON 3 internal . . . 127
4.9 LEON 3 internal with HCODE module . . . 128

5.1 Overview of the proposed architecture . . . 130
5.2 Metadata format description . . . 131
5.3 CCFI-checker state machine . . . 135
5.4 Overhead on code size . . . 138
5.5 Overhead on execution time . . . 139

6.1 Overview of the proposed architecture . . . 144
6.2 Metadata format description . . . 146
6.3 Compilation flow . . . 147
6.4 Exemple of miss branch prediction . . . 149
6.5 Interruption FSM . . . 151

List of Tables

1 Comparaison de certaines des protections les plus importantes 17

1.1 Comparison of some of the most prominent protections 32
1.2 Mapping between CCFI protections and threats 36

5.1 Code and metadata correspondence 141
5.2 Hardware cost 142

6.1 Hardware cost 153
6.2 Hardware cost of caches 153
6.3 Benchmark on A* processor in number of cycle 154
6.4 Software modification 154

Remerciements

Je dédie cette thèse à mon père qui a su me transmettre avec passion le goût de l'informatique, de l'électronique et qui m'a transmis sa soif de connaissance.

Je remercie ma famille pour sa présence, son aide et son indéfectible soutien.

J'aimerai remercier Télécom Paris et Secure-IC qui m'ont accueilli pour réaliser cette thèse CIFRE financée par l'ANRT.

J'aimerai remercier mes directeurs de thèse, Jean-Luc Danger et Sylvain Guilley, qui m'ont donné l'opportunité de faire cette thèse, qui m'ont prodigué de nombreux conseils et ont été présents pour me motiver dans les moments difficiles.

Je remercie aussi mes encadrants, Thibault Porteboeuf et Adrien Facon qui m'ont soutenu durant cette période, avec qui j'ai aimé travailler et qui m'ont beaucoup appris.

Je tiens particulièrement à remercier Karine Heydemann, Ulrich Kühne ainsi qu'Abdelmalek Si merabet pour leur grande aide, leur soutien, leurs conseils avisés et pour tout le temps qu'ils m'ont consacré.

Je remercie tous les enseignants-chercheurs avec qui j'ai eu de nombreuses discussions passionnées et de sympathiques moments de détente.

Je remercie mes amis Céline Brunel, Jérémie Brunel, Boris Ploujoux, Xuan Thuy Ngo, Alexander Schaub, Sébastien Carre, Nicolas Bruneau et Margaux Dugardin pour leur présence et leur soutien dans les moments difficiles et surtout pour les bons moments passés ensemble.

Je remercie mes collègues de Secure-IC qui m'ont aidé en me prodiguant de nombreux conseils techniques, partagé avec moi leur expertise et pour leur bienveillance.

Merci, à toutes ces personnes et à toutes celles que j'aurai pu oublier

Abstract

Summary

In this thesis, we propose a new hardware Control Flow Integrity (CFI) protection implementation called Code and Control Flow integrity. Current CFI protection implementations, either software or hardware, are actually limited to a coarse-grain coverage, or feature an overhead cost which is prohibitive. The latter cost can concern the running time execution or directly impacts the hardware footprint.

We start in chapter 1 by introducing the security problem inherent of today's processors. This security flaw can affect all kinds of systems and constitutes a major problem for system continuity and person safety. The processors variability is discussed in order to understand the field of application of our solution. We present the concept of Control Flow Graph as the basic structure of a program. This allows us to explain on the one hand to what extent CFI is able to protect programs against corruption, and on the other hand what are the advantages and limitations of this protection.

Then we explore in chapter 2 the state-of-the-art of today's protections. This is presented as a list of existing CFI implementations, either software or hardware, academic or industrial. This allows us to distinguish coarse-grain and fine-grain protections, their limitations and advantages as well as their cost. We also propose an overview of historical protections to understand their history and why some protections have been adopted by industrialists while others have been ignored.

Then, in chapter 3, the threat model is exposed to explain the attacker capabilities. This describes what the attacker is able to perform on the targeted processor, and highlights the requirement for efficient protections.

The next three chapters (4 & 5 & 6) present our innovative solution with a top-down approach. We start by setting definition of a program structure

and security property of CCFI. Then, each level of security is detailed, from functions call and return, called inter-procedural, to control flow checking between basic block inside a function, called intra-procedural. The chapter 4 leads to a simple implementation of CCFI and explain each aspect of its security. In the following chapter 5 we describe an implementation deployable on microcontroller and small processor. This presents the minimal hardware implementation needed to offer CCFI protection. The last chapter 6 describes a more advanced implementation, able to deal with complex processors using speculative execution and branch prediction. We also address the mangement of interruptions and propose a flexible solution to prevent false positive and protect code during an interruption.

In chapter 7 we summarize all benefits of the CCFI approach to provide efficient CFI. Its capacity of adaptation on many platforms and its low impact on performance and hardware cost makes it a good candidate to be adopted by the industry. Some perspectives on how this solution can be improved are discussed. It is notably pointed out that CCFI can be implemented in many different ways allowing an adaptation to many different architectures. We also discuss the possibility to use CCFI when an operating system or an hypervisor are used. In this case we show how our solution can be adapted to support context switch.

Contributions

The premise of this thesis was a publication in 2014 at PPREW [25]. These four years of research gave rise to three publications. The first one was published in "The New Codebreakers" book in 2016 [24], the second was in 2018 at DSD conference under the name "CCFI-Cache: A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity" [22], and the last one, "Processor Anchor to Increase the Robustness against Fault Injection and Cyber Attacks", has been accepted for presentation at COSADE [70], in october 2020. This thesis also led to a patent for Secure-IC based on our technique to synchronize metadata and code.

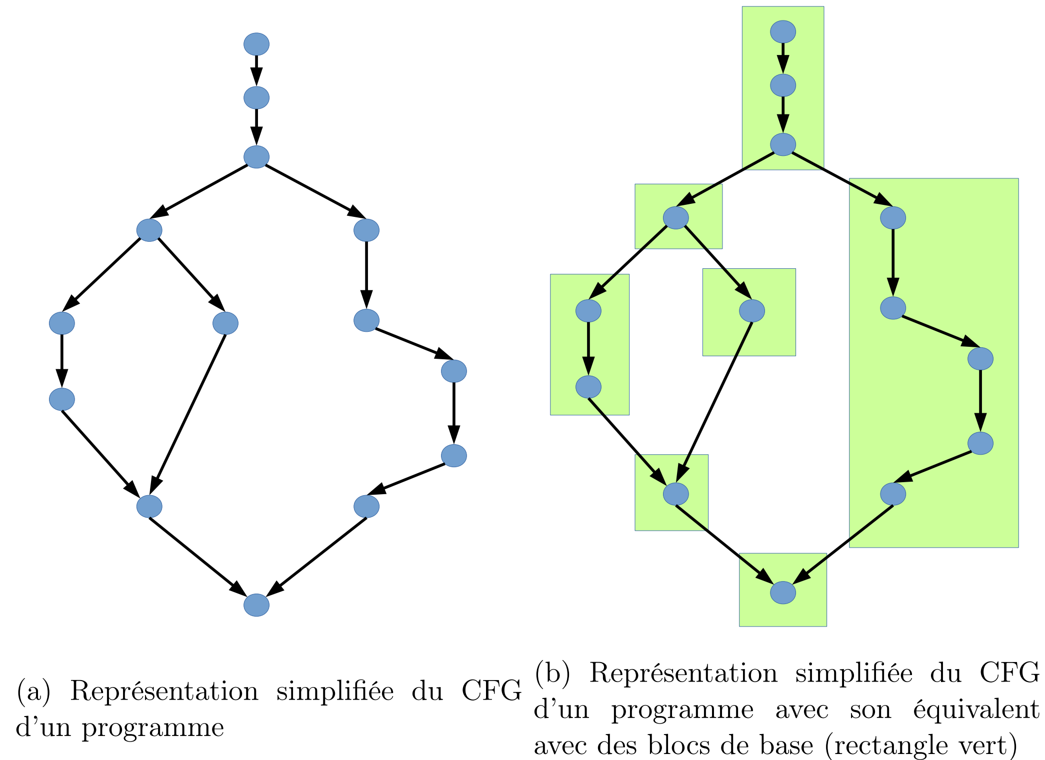
Résumé long en Français

Dans cette thèse, nous proposons une nouvelle implémentation matérielle pour le Contrôle d'Intégrité du Flot d'exécution (CFI: Control Flow Integrity) nommée Code and Control Flow Integrity (CCFI). Les implémentations actuelles, logicielles ou matérielles, implémentant la protection CFI sont actuellement limitées à une protection à gros grain ou ont un surcoût en temps d'exécution ou en terme d'empreinte matérielle trop coûteuse pour être déployée.

Nous commençons par introduire dans le chapitre 1 les problèmes de sécurité inhérents aux processeurs modernes. Ces failles de sécurité peuvent affecter toutes sortes de systèmes et constituent un problème majeur pour la continuité des systèmes et la sécurité des personnes. Les différences entre les processeurs sont discutées pour comprendre les différents champs d'application de notre solution. Nous y présentons le concept Graphe de Flot de Contrôle (CFG: Control Flow Graph).

Un CFG est la représentation graphique du flux d'exécution d'un programme. Les CFG sont principalement utilisés dans l'analyse statique ainsi que dans les applications de compilation. Sur la figure 1a chaque point bleu représente une instruction assembleur et chaque flèche une transition autorisée pendant l'exécution du programme. Afin de simplifier le CFG de la figure 1a, nous pouvons utiliser le principe de Bloc de Base (BB: Basic Bloc) qui définit un ensemble d'instructions assembleurs toujours exécutées ensemble et dont le seul point d'entrée est la première instruction du BB et que le seul point de sortie est la dernière instruction de ce BB. Ainsi nous pouvons réduire le CFG de 14 nœuds à 7 nœuds (voir figure 1b).

Le principal problème de sécurité des processeurs modernes vient du fait que les processeurs n'ont pas connaissance du partitionnement mémoire. Les différentes zones mémoire utilisées par les programmeurs comme la pile, le tas et les pages mémoires de code ou de données ne sont que des constructions faites par les programmeurs afin de faciliter la conception de programme. Mais le processeur n'est pas informé de cette structure, ce qui permet à des attaquants de modifier l'exécution du programme ciblé. Plusieurs exemples sont présentés dans la section 2.1.3. Un exemple significatif est l'attaque de type dépassement de tampon (buffeur overflow) où le programme accède à des données à partir d'un pointeur de tableau mais ces données se trouvent au-delà de la plage mémoire destinée à ce tableau. Ce type de vulnérabilité est présent dans beaucoup de langage bas niveau comme le C.



(a) Représentation simplifiée du CFG d'un programme

(b) Représentation simplifiée du CFG d'un programme avec son équivalent avec des blocs de base (rectangle vert)

Figure 1: Représentation simplifiée du graphe d'un programme

D'autres attaques consistent à directement injecter du code dans l' application. Par exemple, en utilisant un champ de données destinées à contenir des informations de l'utilisateur pour y stocker du code binaire. Ensuite l'attaquant n'a plus qu'à dévier l'exécution du programme sur le code injecté pour finir son attaque. Ce genre d'attaque est maintenant arrêtée par des protections du type `W^X`. Mais cette protection n'est pas présente sur tous les processeurs, en particulier sur les microcontrôleurs. De plus, il existe maintenant des attaques de type réutilisation de code (code reuse) qui permettent d'achever le même type d'attaque sans avoir à injecter le code. Ce qui met à mal les protections de type `W^X`.

Le tableau 1 (pages 17 et 32) résume les différentes protections existantes ainsi que leur niveau de sécurité.

Dans le chapitre 2 nous explorons l'état de l'art des protections existantes. Nous y détaillons la liste des implémentations de la protection CFI,

Table 1: Comparaison de certaines des protections les plus importantes

Protection	$W \oplus X$	SOFIA [27]	Intel CET [38]	ARM PA	PICON [21]	HCODE [23]	PathArmor [71]	HCFI [18]	Our solution
a) Inter-Procedurale	X	✓	✓	✓	✓	X	✓	✓	✓
b) Intra-Procedurale	X	✓	(✓)	X	✓	✓	X	X	✓
c) Intra-BB	X	✓	X	X	X	✓	X	X	✓
d) Intégrité du Code	✓	✓	X	X	X	✓	X	X	✓
e) Non intrusive	X	X	X	X	✓	✓	✓	X	✓
f) Exec. Spéculative	✓	(✓)	✓	✓	✓	X	?	✓	✓
g) Interruption	✓	X	✓	✓	✓	X	?	(✓)	✓

qu'elles soient logicielles ou matérielles, académiques ou industrielles. Cela nous permet de distinguer les protections à gros grain de celles à grain fin et d'en déduire leurs avantages et leurs inconvénients ainsi que leur coût respectif. Nous proposons ainsi une vue d'ensemble de l'historique des protections implémentées dans les systèmes pour comprendre leur histoire et définir pourquoi certaines protections ont été adoptées par l'industrie tandis que d'autres ont été ignorées.

L'état de l'art nous permet de distinguer plusieurs familles de sécurité. La première famille se base sur la gestion de la segmentation de la mémoire. Le principe est de segmenter la mémoire en pages et d'attribuer à chaque page des droits spécifiques. Par exemple des droits d'écriture, de lecture ou d'exécution. Cette technique est poussée plus loin avec les protections implémentant les fat-pointers. L'idée est d'offrir un niveau plus fin de gestion de la mémoire, par exemple être capable de délimiter la zone mémoire pour un tableau.

Le principe de protection des pages mémoire a en premier été implémenté en logiciel mais est maintenant répandu dans tout processeur moderne. Par contre, les techniques de fat-pointer ont encore beaucoup de mal à être déployées en milieu industriel. Principalement en raison de leur surcoût, en terme de mémoire, de temps d'exécution mais également car certaines protections nécessitent des changements au niveau de la chaîne de compilation.

La seconde famille de protections qui nous intéresse est celle implémentant des protections du type CFI. Les implémentations logicielles souffrent d'une vitesse d'exécution fortement réduite empêchant alors toute utilisation concrète. Les implémentations logicielles réussissant à avoir des performances correctes le font en réduisant le niveau de sécurité, par exemple en vérifiant le respect du CFG uniquement au niveau des appels des fonctions. Les implémentations matérielles quant à elles sont généralement plus rapides mais nécessitent une modification du processeur pour fonctionner. Or, dans un contexte industriel, il n'est pas possible de modifier les processeurs, soit parce que l'accès au code source du processeur n'est pas disponible, soit parce que le temps d'intégration et de validation du processeur est trop long.

Le modèle d'attaquant est discuté dans le chapitre 3. Cela décrit ce que l'attaquant est capable de faire sur le système et souligne la nécessité d'une protection efficace.

Nous considérons qu'un attaquant puissant est capable d'exécuter des attaques logicielles et/ou matérielles. En d'autres termes, le vecteur d'attaque peut être soit l'exploitation d'une faille existante (qui permet de corrompre

l'état du programme) ou une altération causée par un stress externe qui modifie l'état interne comme les attaques par injection électromagnétique. Les attaques matérielles sont détectées si la perturbation vise le code ou des pointeurs de code. Notez que certaines attaques récentes (telles que RowHammer et PlunderVolt) sont des modifications matérielles déclenchées par le logiciel et sont donc considérées comme des attaques logicielles et matérielles.

Comme les attaques physiques sont capables de modifier un mot en mémoire sans passer par le gestionnaire de mémoire, elles sont capables de contourner les protections matérielles comme `W^X`. L'attaquant est ainsi en mesure de modifier les instructions ou de sauter une instruction. En tant qu'attaque logicielle, il est capable d'utiliser toute vulnérabilité présente dans le logiciel Cela comprend l'utilisation de dépassement de tampon sur la pile pour modifier l'adresse de retour et de changer des pointeurs de fonction.

Les chapitres 4, 5 & 6 présentent notre solution innovante par une approche top-down. Nous commençons par définir la structure d'un programme et les propriétés de sécurité du CCFI. Chaque niveau de sécurité est détaillé. La protection des appels et des retours des fonctions est appelée inter-procédurale. La protection du flux d'exécution entre les BB est appelée intra-procédurale. Enfin la protection du flux d'exécution au sein du BB est appelée intra-BB.

Comme l'objectif est de fournir une implémentation d'une protection CFI capable de fonctionner sur une grande variété de processeurs, nous voulons que cette dernière ne soit pas intrusive et ne nécessite pas la modification du processeur protégé. Il est aussi nécessaire de prévoir la possibilité de protéger des codes utilisant des mécanismes d'interruptions, car cela peut s'apparenter à une déviation du CFG si cela n'est pas pris en compte. La solution doit aussi être capable de protéger des processeurs plus avancés utilisant potentiellement de l'exécution spéculative. Ainsi nous retrouvons tous les niveaux de sécurité que doit satisfaire notre solution CCFI dans le tableau 1

Le chapitre 4 conduit à une mise en œuvre simple du CCFI et explique chaque aspect de sa sécurité. Dans le chapitre 4, nous décrivons une mise en œuvre sur microcontrôleur et petit processeur. Cela présente le matériel minimal nécessaire pour assurer la protection CCFI.

La figure 2 représente l'architecture de base de la solution CCFI. Elle est assurée par deux modules matériels supplémentaires (indiqué en rouge) : Le `CCFI-cache` récupère les métadonnées qui ont été calculées au moment de la compilation. Ces métadonnées contenant toutes les informations rel-

atives au flot de contrôle. Ces informations sont utilisées au moment de l'exécution par le second module matériel, le `CCFI-checker`. Pour suivre et contrôler l'exécution du processeur, le `CCFI-checker` est connecté aux signaux d'interface entre le processeur et le cache d'instructions.

Le `CCFI-cache` a les mêmes caractéristiques (largeur, taille, associativité, politique de remplacement, etc...) que le cache d'instructions. Pour chaque bloc de base du programme, il y a un bloc de métadonnées correspondant. Chaque bloc de métadonnées est parfaitement aligné en mémoire sur le BB de code correspondant, avec un décalage constant. Pour chaque accès au cache d'instructions, une requête est émise au `CCFI-cache`. Grâce à l'utilisation d'un décalage constant entre les instructions et les métadonnées, le calcul d'adresses complexes est évité. En outre, les données du cache d'instructions et du `CCFI-cache` seront toujours cohérentes, c'est-à-dire que si un BB de code est présent dans le cache d'instructions alors ses métadonnées sont présentes dans le `CCFI-cache`.

Les métadonnées pour chaque BB contiennent trois informations cruciales pour la vérification du flot d'exécution:

- Le nombre d'instructions contenues dans le BB
- Les adresses de destinations valides comme successeurs du BB
- Une signature calculée à partir des instructions et des metadonnées du BB

La vérification proprement dite est réalisée par le `CCFI-checker`. A la fin de chaque BB, il vérifie la validité de l'adresse suivante en la comparant avec les adresses valides contenues dans les métadonnées, assurant ainsi que le CFI intra-procédural est respecté. En cas d'appel ou de retour de fonction, une copie de la pile d'appels (shadow stack) est utilisée pour vérifier le CFI inter-procédural. Cette pile d'appels est intégrée dans le `CCFI-Checker` et n'est pas accessible depuis le processeur principal. La cohérence intra-BB est assurée par un compteur qui contrôle le nombre d'instructions exécutées avant un transfert de contrôle. Enfin, l'intégrité du code et des métadonnées est assurée par une signature pré-calculée stockée dans les metadonnées. Cette signature est comparée à une valeur calculée durant l'exécution du BB par le `CCFI-checker`. En cas de violation, une interruption est déclenchée. Les détails du `CCFI-checker` sont présentés à la section 5.1.3.

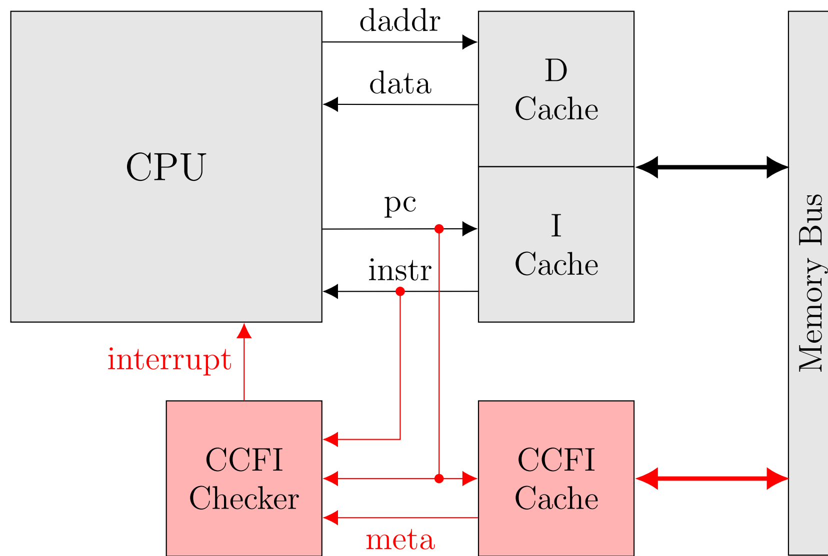


Figure 2: Vue simplifiée de l'architecture de base du `CCFI`

Le dernier chapitre 6 décrit une mise en œuvre plus avancée, capable de protéger des processeurs complexes utilisant l'exécution spéculative et la prédiction de branchement. Nous abordons également la gestion des interruptions et proposons une solution souple pour éviter les faux positifs et protéger le code durant l'exécution du code de l'interruption.

Lorsqu'une interruption se produit, le processeur détermine l'adresse mémoire du vecteur d'interruption et saute dessus. Il y a deux façons principales pour un processeur d'exécuter le gestionnaire d'interruption. La première est d'avoir un code statique en dur dans le processeur, quelle que soit l'interruption il exécutera ce code. La distinction du type d'interruption et l'appel du bon gestionnaire d'interruption sont laissés au programmeur. La seconde est d'avoir une zone de mémoire dédiée à un tableau de pointeurs de code. Pour chaque interruption, le processeur va chercher le pointeur de code correspondant et exécute le code relatif. Dans tous les cas, il en résulte un saut direct à la sous-routine d'interruption à tout moment et de n'importe où. Ce comportement, vu de l'extérieur, est considéré comme une violation du CFG. Pour être agnostique sur le type d'interruption, notre solution consiste à ajouter un drapeau `Int` dans l'en-tête des métadonnées du premier et du dernier BB de la fonction du gestionnaire afin de détecter le début et la fin de l'interruption.

Cela permet de détecter les interruptions déclenchées à la volée, indépendamment de la la mise en œuvre du processeur. En cas d'interruption, le `CCFI-checker` détecte la discontinuité du flux de contrôle, mais les métadonnées indiqueront que la destination est à la fois un `Start BB` et `Int` signifiant que cette fonction est appelée à cause d'une interruption. Dans ce cas le `CCFI-Checker` sauvegarde son contexte d'exécution actuel dans une mémoire interne et la shadow stack est utilisée pour sauvegarder le PC actuel. Une fois la sauvegarde du contexte fait le `CCFI-Checker` saute à un autre FSM dédié à suivre l'exécution du gestionnaire d'interruption. Cette machine à états est la même que l'état normal, sauf pour l'état `END BB` où le drapeau `ENDINT` est vérifié. Si l'indicateur `ENDINT` est présent, alors le précédent contexte sauvegardé est restauré. Une fois le contexte restauré, le `CCFI-checker` peut poursuivre la vérification du BB là où elle a été interrompue.

Pour réussir à protéger les processeurs utilisant de l'exécution spéculative, il faut faire la distinction entre l'instruction récupérée par le processeur et l'instruction exécutée. Pour cela chaque instruction extraite est stockée dans un tampon circulaire avec ses métadonnées correspondantes. Ce tampon circulaire est aussi profond que nécessaire pour reproduire la latence du pipeline entre l'étage de la collecte (fetch) et l'étage de l'interface de traçabilité (voir figure 3). Pour chaque instruction récupérée par le processeur, une ligne est stockée dans le module `Buf`. Chaque ligne dans le module `Buf` contient l'instruction, son adresse et les métadonnées associées. Ainsi, lorsque la sortie de l'interface de trace expose une adresse, cette adresse est envoyée au module `Buf` pour sélectionner l'entrée correspondante. Le module `Trace Decoder` est présent pour assurer la récupération de l'adresse du PC à partir de l'interface de trace. Par construction, il n'est jamais possible pour le processeur de sortir un PC à partir de l'interface de trace sans que les métadonnées correspondantes soient présentes dans le module `Buf`. Une fois la ligne sélectionnée, le module `Buf` envoie les informations (PC, instruction et les métadonnées) au `CCFI-Checker`. Ce faisant, le `CCFI-Checker` est en mesure de suivre l'exécution du processeur, étape par étape, sans erreur, et cela même avec un processeur utilisant l'exécution spéculative.

Dans le chapitre 7, nous résumons tous les avantages de l'approche du CCFI pour fournir un CFI efficace. Sa capacité d'adaptation sur de nombreuses plateformes et son faible impact sur les performances et le coût matériel, en font un bon candidat à l'adoption par l'industrie. Quelques perspectives sur la manière dont cette solution peut être améliorée sont discutées. Il est notamment souligné que le CCFI peut être adapté de différentes

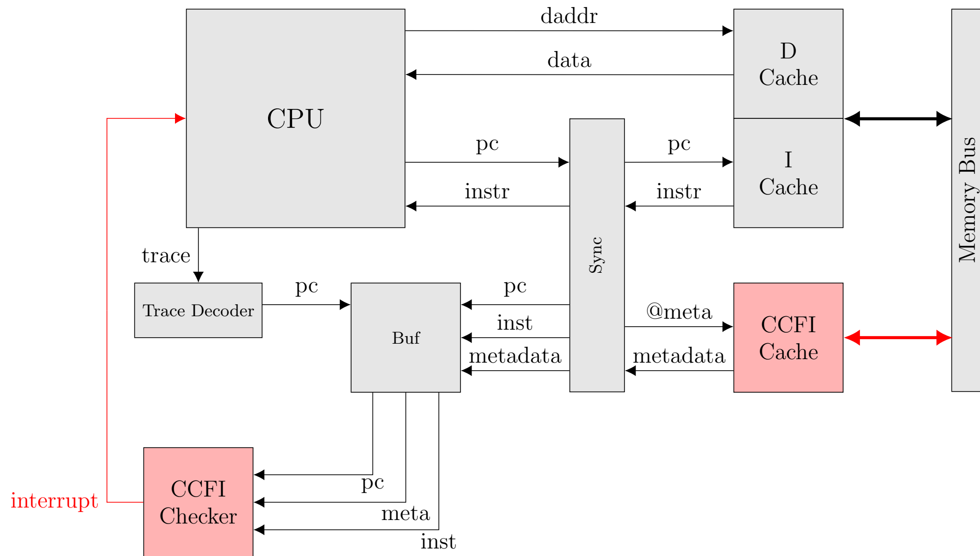


Figure 3: Vue simplifiée de l'architecture de la solution `CCFI` prenant en compte l'exécution spéculative

manières afin de permettre une adaptation à des architectures différentes. Nous discutons également de la possibilité d'utiliser le CCFI lorsqu'un système d'exploitation est utilisé. Dans ce cas, nous montrons comment notre solution peut être adaptée pour soutenir le changement de contexte et les modifications à apporter au système d'exploitation.

La vérification de l'intégrité du code et du flux de contrôle en parallèle avec l'exécution du logiciel est de la plus haute importance pour les applications de sécurité et de sûreté. L'idée de base d'utiliser un cache pour récupérer les métadonnées en même temps que le code a montré son efficacité en terme de performance, sans pour autant diminuer le niveau de sécurité. En effet, l'alignement des métadonnées avec la structure du code a été une bonne idée, tant pour simplifier le calcul de l'emplacement des métadonnées à partir de l'adresse du code que pour utiliser au mieux le cache de métadonnées. Nous avons prouvé qu'il est possible de développer une solution de CFI à grain fin tout en maintenant de bonnes performances. Un avantage majeur de notre solution est sa taille réduite en coût silicium et indépendante de la puissance de calcul du processeur utilisé.

Néanmoins, certaines IPs spécialisées sont nécessaires en plus de la conception de base pour travailler avec les processeurs complexes, par exemple

ceux qui utilisent l'exécution spéculative. Le surcoût mémoire lié à l'ajout des métadonnées, doublant la taille mémoire nécessaire pour le code, peuvent faire l'objet de controverses parmi les industriels et les universitaires. Cependant, nous pouvont considérer que c'est un prix raisonnable à payer pour ce niveau de sécurité et qu'aujourd'hui la mémoire est relativement bon marché. En outre, lorsque le code est compilé efficacement, il reste dans le cache et donc la vitesse d'exécution n'est pas impactée. Dans d'autres cas d'utilisation, doubler la taille mémoire est difficile surtout lorsque cette mémoire est intégrée dans un SoC (System-On-Chip). Comme nous l'avons vu, lors du développement de la solution CCFI, un des défis principal est de pouvoir détecter le début et la fin des blocs de base (BB) au cours de l'exécution. En effet, lors de l'exécution, seules les instructions qui modifient le flux de contrôle donnent un aperçu du CFG, mais cela ne suffit pas pour déduire l'ensemble du CFG. Nous avons intelligemment contourné ce problème en ajoutant ces informations directement dans nos métadonnées. Nous soulignons que la solution garantit le CFI au niveau des BB, ce qui le rend efficace et capable de détecter toutes les attaques modifiant le déroulement de l'exécution du programme. L'impact de la solution sur les performances est prometteur avec une variation de 2 à 30% selon le type de programme et le type d'architecture mis en place, ce qui est très raisonnable et industriellement viable.

Ayant démontré son efficacité, la solution CCFI ouvre toutefois un certain nombre de perspectives en terme d'amélioration. La première piste d'amélioration est de lever la restriction forte de l'implémentation actuelle qui requiert que les métadonnées soient alignées avec le code. Ce qui implique que les métadonnées requièrent autant de place que le code. Or une grande partie des métadonnées ne contiennent pas d'informations utiles et certains BB du code sont artificiellement agrandis ce qui impacte les performances du programme. Il serait donc intéressant d'explorer la possibilité d'assouplir cette contrainte afin de réduire l'empreinte mémoire et d'améliorer les performances. Toute la difficulté étant de trouver un algorithme qui permet de déterminer l'adresse des métadonnées aussi rapidement que possible. Une possibilité serait de stocker l'adresse des métadonnées du BB suivant à côté de l'adresse du code au sein même des métadonnées. Nous pouvons aussi explorer la possibilité d'améliorer les performances de façon globale en modifiant le compilateur pour générer des BB de base suffisamment grands pour contenir toutes les métadonnées au prix, par exemple, de dupliquer du code. Une dernière optimisation concerne le `CCFI-cache` que nous pouvons intégrer

directement au cache d'instructions pour alléger le nombre de requêtes sur le bus de mémoire et factoriser la logique du `CCFI-cache` avec celle du cache d'instructions.

Chapter 1

Introduction

1.1 Today's processors

1.1.1 Common architecture of today's processors

Today processor are everywhere, inside computer, smartphone, smartwatch, calculator, TV box, toys and a lot more with the IOT devices. For all these purposes different kinds of processors have been developed from small microcontroller able to operate at few dozen of Mhz to powerful processor containing multiple cores, each operating at over 3.0 Ghz. This gap of performance is due to architectural differences, while microcontrollers have only a 3 stage pipeline and limited caches or none at all, big processors have around 14 stage pipeline and large caches. Advanced processors also have advanced prediction system to guess which part of the program while be executed. This in order to keep the pipeline full and to keep the number of instructions executed per cycle high. This complexity comes with a physical size and cost increase.

While it is relatively easy to follow the execution of a program from the outside of the processor by looking at which address it fetch instruction. It is more difficult, event impossible, to follow the execution path of the program for more complex processors. This can be because of prefetch mechanism which add a hidden cache inside the processor to speculative execution. During speculative execution the processor will guess the next jump before it had computes the destination. If this guessing was right it is time saved but in case of miss prediction the processor will make an internal rollback to previous state and take the right one. From the outside of the processor it

is hazardous to guess if the processor has make a wrong prediction or if it was an unexpected behavior. This can be theoretically possible if we know exactly the internal functioning of of the prediction mechanism. But this is rarely the case for most of the processor. It would come back to recreate a part of the processor to be able to follow processor execution. This will become more and more expensive as the processor have complex and multiple predictions.

1.1.2 RISC Processors

CISC processors can have a large number of possible instructions, to minimize the code size and its impact on the necessary size of the instruction cache, these instructions have various length. This particularity makes harder to detect the beginning of an instruction in memory. It's also a risk for the security because instruction can be misinterpreted if they are fetched with one byte of offset. RISC processors feature constant size instruction which facilitates code instrumentation and lower the complexity of their pipeline. This is easier to detect misaligned fetch and to detect the beginning of an instruction. Their execution is also much more deterministic, for these reasons we focus on RISC processor in the rest of this PhD.

1.2 Control Flow Graph

At the level of the machine code, a program is composed of multiple functions which in turn can be decomposed into basic blocks. A *Basic Block* (BB) is a straight-line sequence of instructions with a unique entry point and a unique exit point, i.e. if the control flow enters a BB, it will execute all of its instructions in sequence until leaving the BB at the exit point. A control flow transfer can only take place at the last instruction. This means that a basic block begins by an instruction that is the destination of a jump instruction (typically: branch, jump, call or return), and finishes by a jump instruction (idem: branch, jump, call or return) or by an instruction whose next instruction is the beginning of a new basic block. Each function can be represented as a *control-flow graph* (CFG), where each node corresponds to a BB, and edges represent the control transfers between the BBs. For example, Figure 1.1 illustrates a program along with its flow graph. These CFGs of function always have only one BB to enter the function and only

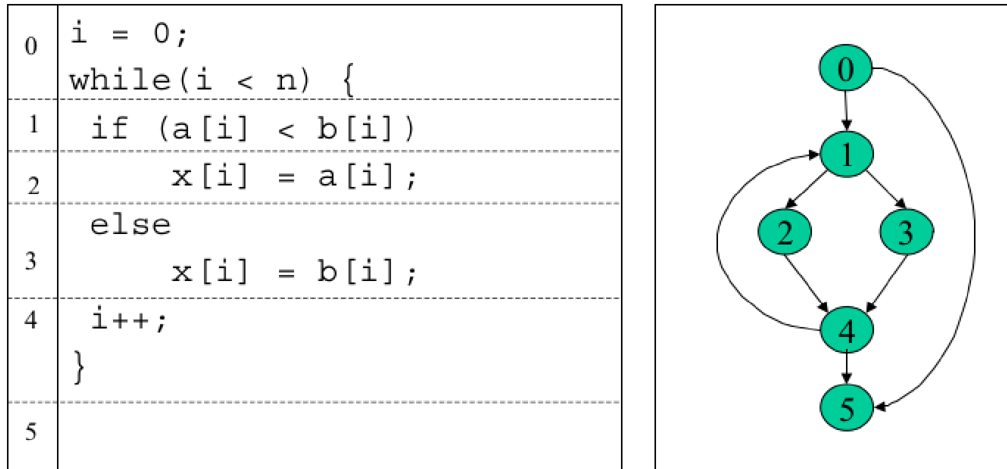


Figure 1.1: Example of program and associated flow graph (from [34])

one BB to exit the function. This constraint is set by the compiler during compilation. A whole program is composed of the CFG of each function linked by edges representing function calls and returns. It is noted that it is common in compilation world to consider that a BB can contain a call to a function without ended it. In our case we consider a call automatically set the end of the BB.

During a nominal program execution the flow remains on the CFG. However due to no memory safety of low level language it can happen that the program executes outside of the CFG. This behavior can be caused by software bug or physical disruption. Such behavior can be exploited by an attacker to leak date or take control over the machine. It is therefor extremely important to guarantee execution integrity. Technique to perform are referred to as Control Flow Integrity.

1.3 Control Flow Hijacking

There are multiple ways to an attacker can hijack the program execution to take over the machine. In many cases, buffer overflows – due to bad programming – offer an entry point for an attacker. They can be exploited to inject code and/or to compromise return addresses stored on the stack to divert the execution flow.

Executing injected code can be mitigated by DEP which prevents written data to be executed. This protection can be circumvented by *code reuse attacks* that rely on (stubs of) existing functions in libraries, so-called *gadgets*. Known variants of this type of attacks are *return-oriented programming* (ROP), *jump-oriented programming* (JOP) or *call-oriented programming* (COP) [17]. As shown in [68], all such attacks rely on code pointer corruption. In this way, only legitimate code of the application is executed, but the CFG of the program is not respected anymore. Executing injected code can be mitigated by Data Execution Prevention (DEP)/W^X which prevents written data to be executed. This protection can be circumvented by *code reuse attacks* that rely on (stubs of) existing functions in libraries, so-called *gadgets*. Known variants of this type of attacks are *return-oriented programming* (ROP), *jump-oriented programming* (JOP) or *call-oriented programming* (COP) [17]. As shown in [68], all such attacks rely on code pointer corruption. In this way, only legitimate code of the application is executed, but the CFG of the program is not respected anymore.

Another threat – invalidating DEP protections – are fault attacks, where memory contents are altered by physical means [43, 73]. Using the *RowHammer* attack [44], a dynamic RAM cell can be changed by rapidly reading neighboring cells before a refresh. Its stealthiness makes this threat extremely dangerous: Even trusted firmware code with a digital signature can be corrupted when residing in RAM. Some examples of attacks enabled by such modifications are Shamir's bug attack [12] (e.g., on RSA) or Sbox tampering attacks [6] (e.g., on AES). Fault attacks are difficult to master, but can be used to change instructions, to manipulate access rights, to skip an instruction, or to directly change the current program counter, in some cases without violating the CFG.

1.4 Control Flow Integrity

Control-Flow Integrity (CFI) refers to protections against control-flow hijacking and was introduced in Abadi's [3] [4] seminal papers. In the paper of 2005 Abadi et al. present a software implementation of CFI. In theory CFI limit all control-flow transfers to those who are intended by the programmer. This prevent the use of code injection, ROP and JOP. CFI is divided in 2 phases, first phase is an analysis phase of the source code to extract the CFG of the programme. This phase can be done on the generated assembly code

but it will be less precise due to the fact we lose information on indirect jump. This is why it is preferable to directly extract the CFG from the source code. The second phase is an enforcement of the CFI by checking the current path of execution against the CFG data extracted during the previous phase.

To prevent control flow hijacking and fault attacks, it is necessary to ensure that control transfer instructions execute as expected, i.e. any control transfer originates from an address that corresponds to a control transfer instruction and targets a valid destination address for this specific instruction. For direct jumps and conditional branches, the valid destinations can be determined at compile-time. Verifying the integrity of these control transfers boils down to checking that for each executed jump or branch, there is a corresponding edge in the function's CFG. We refer to this check as *intra-procedural CFI*.

A different treatment is needed for function calls and returns. Since common functions – such as `printf` – are called from many sites, just checking that the function returns to one of these call sites does not provide a reasonable protection against ROP attacks. Instead, the correct pairing of call and return addresses needs to be ensured. We refer to this as *inter-procedural CFI*.

It should be noted that indirect jumps and calls pose a special problem for CFI as the set of destination addresses can be significant. However, in many cases – such as a `switch` statement which has been compiled to an indirect jump – the set of target addresses is usually small and can often be determined at compile-time.

Otherwise, either manual code changes are necessary or these specific instructions must remain unprotected.

While these checks only consider control transfer instructions, it is necessary to ensure that *inside a BB*, all instructions are executed in-order, thereby preventing instruction skips. This verification, which is hard to implement in software, is called *intra-BB CFI*. Finally, *Code integrity* (CI) refers to verifying that all instructions have been executed *unaltered*. This prevent physical injection to to modify an instruction in memory to change the result of computation, to bypass some verification.

In summary, a combined CFI and CI protection must ensure basic block integrity and verify both intra-procedural and inter-procedural control transfers.

Table 1.1: Comparison of some of the most prominent protections

Protection	W⊕X	SOFIA [27]	Intel CET [38]	ARM PA	PICON [21]	HCODE [23]	PathArmor [71]	HCFI [18]	Our solution
a) Inter Procedural	✗	✓	✓	✓	✓	✗	✓	✓	✓
b) Intra Procedural	✗	✓	(✓)	✗	✓	✓	✗	✗	✓
c) Intra BB	✗	✓	✗	✗	✗	✓	✗	✗	✓
d) Code Integrity	✓	✓	✗	✗	✗	✓	✗	✗	✓
e) Non-intrusive	✗	✗	✗	✗	✓	✓	✓	✗	✓
f) Speculative Exec.	✓	(✓)	✓	✓	✓	✗	?	✓	✓
g) Interruption	✓	✗	✓	✓	✓	✗	?	(✓)	✓

1.5 Introduction

Software are know to contains unsolicited behavior called bugs. These bugs are present in all kinds of systems ranging from personal computer and cell-phone to cloud-servers via industrial control systems. This bug can be a risk for people, environment and also for infrastructures. In fact these bugs are exploited by malicious attackers to extract sensitive data or to take control over the device. While these vulnerabilities can be eliminated by using technique or tools like technical code review, static analysis, usage of memory-safe programming languages or mechanically proved programs, in reality it is difficult to do. This can be due to the complexity and the time needed to use expert tools to prove lack of vulnerabilities. There are also many already existing programs use today which have been written long time ago that have vulnerabilities. One solution would be to rewrite these code in memory safe language but the investment in cost in time is too high the fault to by time-to-market constraints, legacy code reuse, and high competition.

Cyber-attacks exploit these bugs in different ways. There is another threat that can overcome some protection: fault injection. The software attacks exploit bugs or wrong configurations in order to hijack the control flow. In practice, such cyber-attacks mainly operate in two distinct manners. Attacks such as Return-Oriented-Programming (ROP) consist in corrupting the stack such that it calls carefully picked pieces of code called gadgets, which altogether form the attack payload. ROP attacks are thus “code-based”. The second type of cyber-attacks exploit contamination of data to force pointers to different locations. Such “data-based” attacks exploit improperly checked user-inputs, which can lead to control-flow contamination. In this article, we focus on “code-based” attacks, including their protection.

Control-flow integrity (CFI) refers to protections against control-flow hijacking and was introduced in Abadi’s seminal paper [4]. The idea is to verify at run-time by a monitor process or by dedicated hardware that the correct control flow is respected. A common specification of the control flow is given by the static *control flow graph* (CFG) of the application, which can be determined at compile-time.

Since [4] was published many CFI implementation was proposed. There is two main approaches to implement the CFI, software or hardware. Software CFI solutions are convenient mainly because they can be deployed on existing equipment. These solutions rely on instrumenting the software to add self verification or by using an external monitor to check the behavior of the

monitored thread. The flexibility of the software solution is at the cost of performance slow down or to only ensure coarse grained CFI, like [21]. On the another hand, hardware solutions are generally proposed in academic papers and rely on hardware monitor to follow the execution of the processor. Or by using cryptographic primitive with core modification to ensure CFI and more. These hardware solution have generally less impact on processor performance but needs modification of the internal of the processor, which is a very high price to pay in the industrial world, like SOFIA implementation [27].

Most CFI approaches assume that the code cannot be modified, due to the presence of widely used *data execution prevention* (DEP) protections. Such a protection is commonly present on high performance processors but rarely deployed on embedded platforms or micro-controllers. Furthermore, different threats may invalidate this assumption: There exist physical attacks able to perform fault injections that result in a modification of the executed code [43, 73]. Since the discovery of the *RowHammer* attack [44], it is known that changes in write protected DRAM can even be induced by software. Hence, *code integrity* (CI) is also to be targeted in order to protect systems against a large body of attacks that disrupt the execution.

Hardware-based solutions range from lightweight solutions – ensuring only some types of control transfer (such as a so-called *shadow stack*) or reducing the amount of reusable code by marking valid call/jump destinations – to solutions covering all control transfers that can be determined statically at compile-time, at link-time, or at load-time of the application [23, 27]. Unfortunately, such approaches either offer coarse-grained protection or does not allow indirect jump or they require a significant modification of the CPU, which prevents them from being deployed in practice due to either the huge amount of work required for validating a modified processor or the use of off-the-shelf processor cores. This is why we target a *non-intrusive* solution that does not modify the CPU core.

Another technical point generally not addressed by CFI implementation is how to handle interruptions. Interruption can happen at any time, and can be seen as a violation of the CFG from an external point of view. It is also necessary that the hardware implementation of CFI addresses the speculative execution or branch prediction. This adds another complexity level to the CFG verification as unused predicted instruction should not be checked by CFI.

We present a hardware-based solution that combines CFI with CI, while being non-intrusive. The *Code and Control-Flow Integrity* (CCFI) checks are

performed at runtime by a dedicated hardware module outside the processor core. The control flow information – referred to as *metadata* – is stored in a dedicated section in memory and is aligned with the instructions. These metadata are fetched by a cache named CCFI-cache. Whenever a new instruction is requested by the processor, the corresponding metadata is fetched transparently and in parallel, so as not to disrupt or slow down the execution flow. The *CCFI-checker* verifies the integrity of execution flow changes by checking the effective target addresses. Function calls and returns are protected by an integrated shadow stack. Additionally, we ensure code and metadata integrity by computing a signature based on the executed instructions and metadata fetched by validating it against a precomputed signature contained in the metadata.

The proposed *CCFI* solution architecture has been implemented on a RISC-V [11] platform, without modifying the processor core and without adding sanity check code within the program. Our experiments show that the run-time overhead is acceptable for different benchmarks. The price to pay for this very flexible solution is a two-fold increase in instruction memory.

In summary, the contribution of this work is a novel hardware-based CFI scheme that:

- is non-intrusive, since the CPU core remains untouched,
- implemented code integrity (CI),
- is fine-grained, in that it enforces intra- and inter-procedural CFI,
- has low run-time overhead, and
- only requires very minor code modifications from the application code side.

The aforementioned protections, namely code integrity check and intra- and inter-procedural CFI, address hardware and software threats, to various extents, as depicted in Tab. 1.2. The code integrity is granted by various system-level mechanisms already (`W^X`, etc.), but the code is very fragile with respect to hardware-level modifications. On the contrary, control flow hijacking can be achieved more reliabily by the exploitation of software issues. A complete analysis is provided in chapter 3.

Table 1.2: Mapping between CCFI protections and threats

	Hardware attack	Software attack
Code Integrity	+++	+
Control Flow Integrity	+	+++

Chapter 2

State-of-the-art

2.1 Software weaknesses and associated attacks

2.1.1 Introduction

This section first introduces the general structure of a program. Particularly, in the case of Executable Link Format (ELF) format, the static and dynamic structures are described in detail, followed by the vulnerabilities induced by the use of of these representations in memory and how they can be exploited. Finally, different methods for payload creation are reviewed.

2.1.2 Program architecture

This section is about the general structure of a program in memory, although some aspects specific to the ELF (Executable Linkable Format) format are mentioned. This format is used by numerous systems, especially GNU/Linux systems (embedded or not). The ELF format specifications are openly available ([20]).

First we present static structures which are declared at compilation time and are allocated inside the ELF file. Secondly we introduce dynamic structures allocated during the execution of the program. Finally we introduce dynamic linking mechanisms for libraries.

Static structures

The structure of a program in memory is divided into sections, in order to make memory management easier for the programmer. It is mainly composed of the following sections:

- `.text` containing the program instructions.
- `.rodata` containing read-only data.
- `.data` containing the initialized data, which is accessible for reading and writing.
- `.bss` containing non initialized data, or data initialized with 0, which is also accessible for reading and writing.

These sections allow to sort the different types of information in memory. For example, the `.text` and `.rodata` sections can be read from a non-volatile memory, for example a Flash memory, and the `.data` section can be copied into Random Access Memory (RAM) to increase performance.

Additionally, some ELF binaries are endowed with `.init`, `.fini` `.ctors` and `.dtors` sections, which serve to call initialization code (`.init`, `.ctors`) or termination code at the program exit (`.fini`, `.dtors`).

It is important to note that microprocessors have no knowledge of the semantic that the developer choose to assign to each of the sections. To be more precise, the microprocessor does not even consider the notion of section at hardware level, as it is actually a generic calculation module. The data semantic and the breakdown into sections are defined by the developer for convenience. The attacks introduced later in this document are linked to the genericity property of the microprocessor, viewed as a general execution module.

It is interesting to note that the control of sections such as `.text`, `.fini` or `.dtors` may give an attacker the control of the instruction flow during the execution or at the program termination. These sections are therefore sensitive areas. Also, the possibility to execute code from the `.data` section is an ideal attack mean, as it is explained in the next sections.

After this introduction about the main big "static" structures of a program, the next section covers "dynamic" structures, namely the heap and the stack.

Dynamic structures

The heap and stack are also stored in RAM. These structures allow the user to perform dynamic allocation during the program execution, unlike the others sections previously presented which are statically allocated (even though their content changes dynamically).

Function calls and stack: For programming flexiblity, programs are organized in functions. This allows to reuse code at several locations in the program, and also to use recursive procedures.

Each function needs resources to be functional, particularly local variables. Moreover, it is necessary to store the parameters of a function call, as well as the state of the registers of a calling function in order to resume the previous computation after the called function was correctly executed.

The microprocessor registers, being limited in number (16 on Intel x86 architectures), dynamic allocation is necessary to satisfy the memory needs of functions. To store all these data (local variables, registers, parameters and return address), we use a stack structure. This stack is a dynamic structure allocated during the execution of program. Each time a function is called, it is used to save the current state of registers and store the address where to return once the called function has finished. A function can also store their local variables on the stack to be able to manage more data than the number of available registers.

At hardware level, the stack is generally addressed by at least one register which serves to identify its top position, as seen in the microprocessor address space.

Function call conventions, and therefore the organisation of data on the stack, may vary depending on the considered microprocessor architecture, the compiler, and the Operating System. For example, some microprocessors have both a register to store the stack current top position, called `stack pointer`, and a register to save the stack top position before the last function call. This register is generally called `frame pointer` and is frequently used as a reference to locate data on the stack more easily, in particular the function call parameters. A frame is a memory section of the stack where data of the current function is stored. In a normal behavior the function must not access to other sections of the stack. `frame pointer` is used to access local variables more easily than by using the `stack pointer`.

Figure 2.1 (page 40) shows how the various items are saved on the stack

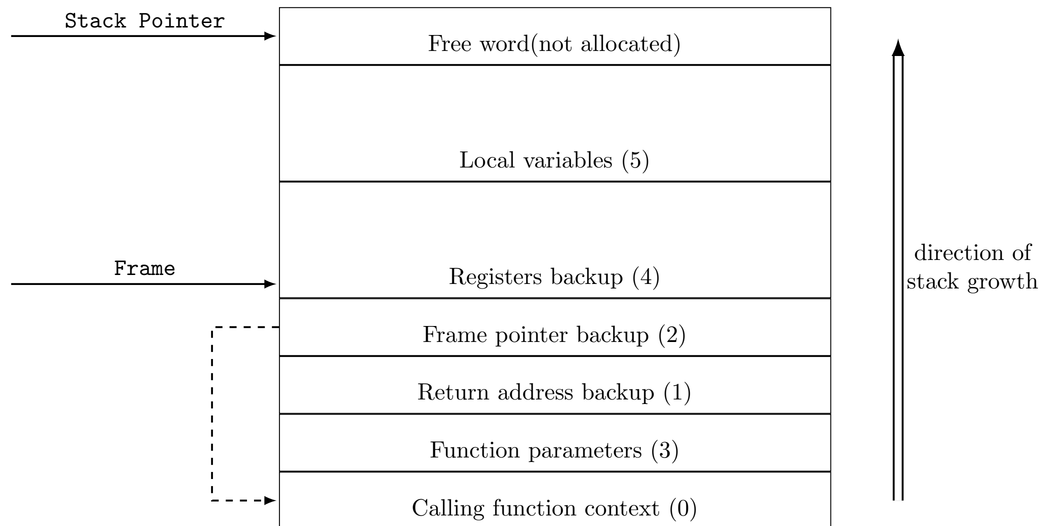


Figure 2.1: Typical data structure on the stack

when a function is called, including at least a return address (1) to get back to the calling function, and optionally the `frame pointer` (2). The function arguments (3) can be passed partially by the registers (e.g. on SPARC architecture) or fully on the stack. Finally, the called function starts its execution by saving the processor registers that will be used (4), and by allocating the necessary space for local variables (5) that cannot be saved into registers. These registers used by the function are then restored to their original state before resuming the calling function.

When the function call returns, local variables are freed by incrementing the `stack pointer`, and the registers are then restored from the backups before those are also freed. At this point, the `stack pointer` has the same value as the `frame pointer`, which is restored from its backup on the stack. Finally, the return address backup is loaded, then freed on the stack, before being used to return to the calling code.

Quite a few sensitive points can already be identified about this dynamic structure. The registers backups and the function call arguments are potentially important with regards to security, but more importantly, the saved return address, also called *code pointer*, is an address that references code.

This variable is extremely important because its purpose is to modify the execution flow of the program. Taking control of this return address allows

the attacker to take control of the execution flow when the called function ends. This vulnerability (or architectural flaw) is presented in greater detail in the section 2.1.4. The control of the `frame pointer` backup allows the attacker to control what the calling function considers the "stack context" (denoted (0) on the figure) once the called function ends. This includes arguments, local variables, return address of the calling function, and so on.

Heap: Like the stack, a heap is a structure for dynamic memory allocation, commonly present in a lot of programs. This structure is not necessarily used in critical softwares or in embedded systems with low complexity, which rely mainly on static allocation and on the stack. However, it can be found in the vast majority of other programs, especially in Operating Systems such as Windows, GNU/Linux, etc.

The heap is allocated and deallocated by two types of function calls. The first function call (e.g. `malloc` for C/Unix) requests a memory allocation for an area of a given size `n`, which can be a variable of the program. The second function call (e.g. `free` for C/Unix) allows to deallocate a memory area previously allocated.

Several mechanisms may be involved when allocation and deallocation are requested by the user, depending on the size of the concerned memory area. Accordingly to the exploitation methods presented in the next sections, we will consider memory allocations of small memory areas. Indeed, these areas are the objects of specific procedures from the heap manager, which is an opportunity for the attacker.

Thereafter, the focus will be put on the `dlmalloc` allocator, formerly present in systems equipped with GNU LibC. This allocator has been replaced by `ptmalloc` which, while respecting the general principles of `dlmalloc`, introduces consistency tests to detect potential memory corruptions, as well as mechanisms for concurrent accesses in *multi-threaded* systems.

dlmalloc memory allocator The heap manages allocation and deallocation requests by splitting and handling a dedicated range of memory, whose size may vary depending on the memory requirements of the program.

This memory zone is segmented into pieces called *chunks*. The *chunks* fit into several categories according to their size. Each *chunk* contain the necessary information to the proper management of the heap (e.g. its size, the position of the next *chunk*, etc.) and a zone dedicated to user data (see

Figure 2.2).

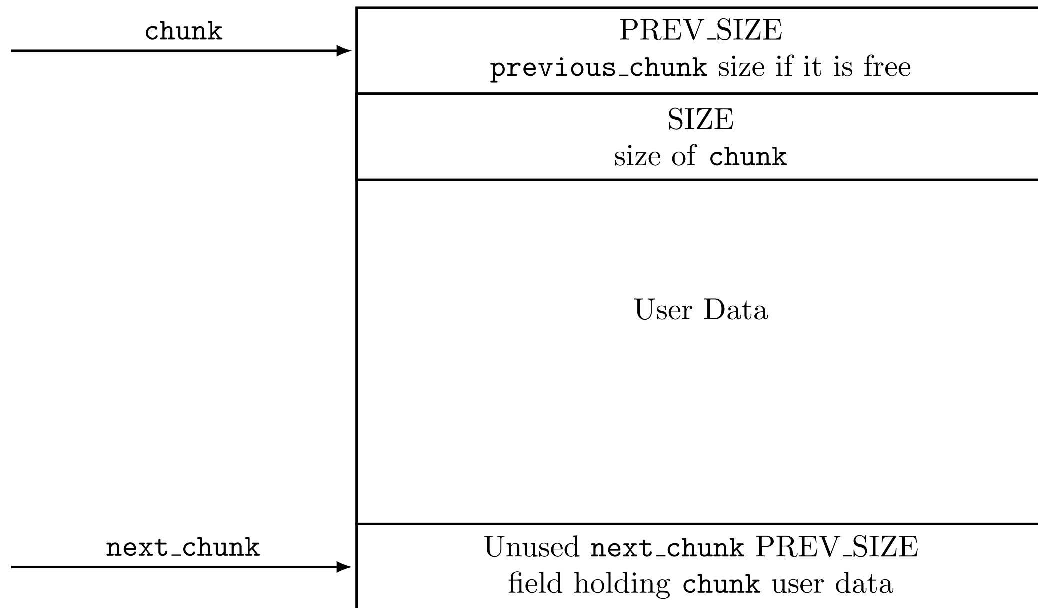


Figure 2.2: Structure of an allocated memory *chunk*

The structure of the *chunk* and its metadata vary depending on its state: free or allocated. The last *chunk*, called *wilderness chunk*, is special and borders the memory space allocated to the heap. This *wilderness chunk* size can be increased or reduced depending on the memory requirements.

When a *chunk* is freed, the manager first checks if it is adjacent to another freed *chunk*. If so, the two *chunks* are merged. If not, the field PREV_SIZE of the next *chunk* is initialized, and the first two words of the area dedicated to the current *chunk* user are used to save the pointers `fd` and `bk`, which will serve to insert the *chunk* in the doubly linked list of free *chunks* (see Figure 2.3).

Two free *chunks* cannot be positioned one after another as the `dlmalloc` manager would merge them. Consequently, such a situation would indicate a heap corruption. `dlmalloc` exploits this property to optimize the memory utilisation by using the PREV_SIZE field of the following *chunk* to store the user data of the current *chunk*.

The heap contains several of these linked lists, called "bins". Each of these lists corresponds to a size or a group of sizes of *chunks*. Thus, when a new *chunk* is being allocated, the lists can be used to search a *chunk* of

size greater or equal to the requested space, in order to reuse it. In Figure 2.4, the *chunk* is therefore inserted by the `frontlink()` macro into a list of *chunks* as presented in the Figure 2.3 page 43.

When a *chunk* is allocated again or when it is merged with another recently freed *chunk*, it is removed from the linked list via the `unlink()` macro.

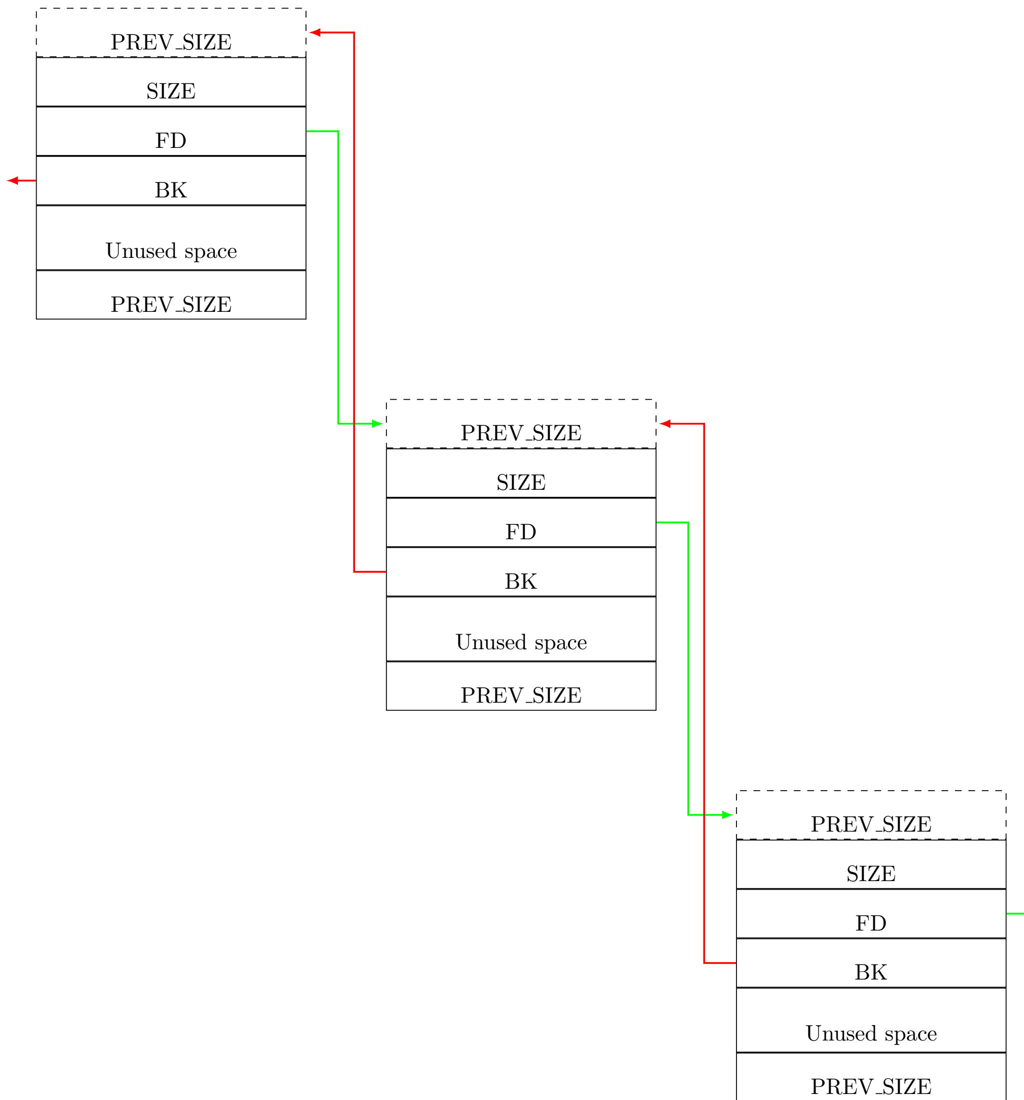


Figure 2.3: Linked list of “free” memory *chunks*

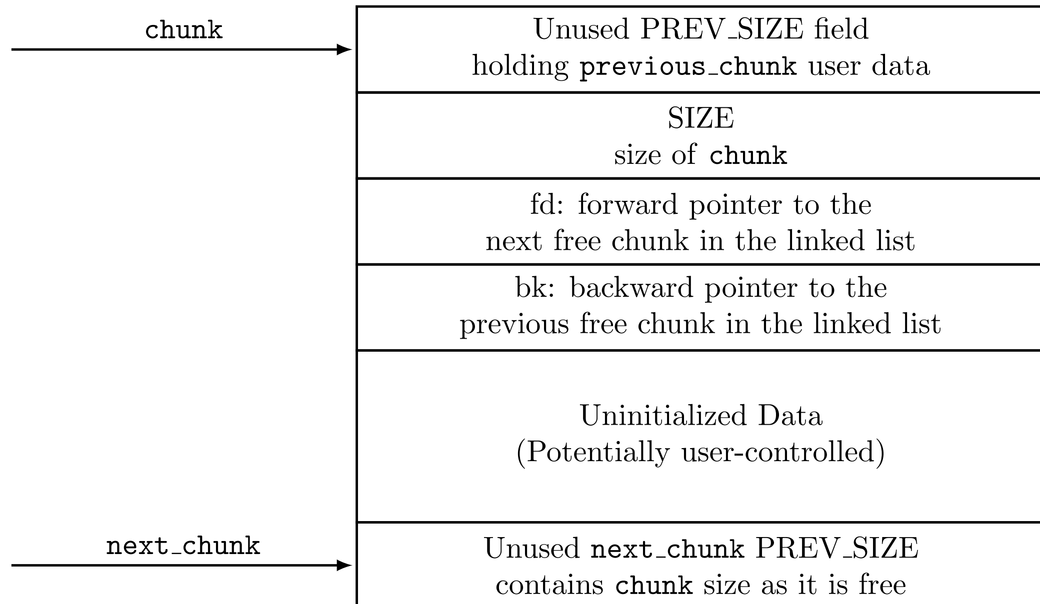


Figure 2.4: Structure of a "free" memory *chunk*

It will later be explained that if an attacker controls the metadata of one or many *chunks*, it can lead to exploitations. Finally, the status of a *chunk* (allocated or free) is stored with a flag in the field SIZE, of which the two least significant bits are unused, given that allocations are aligned on 32-bit words.

Metadata corruption may lead an attacker to write data in memory at arbitrary addresses, or to control a program memory, which makes metadata sensitive data.

Dynamic Loading of Libraries.

The ELF (see 2.1.2) standard specifies three different types of files format, each with a precise role in the construction chain of an executable program in memory. Prior to the execution, three phases can be distinguished, represented in Figure 2.5 (page 45):

1. The compilation: this step converts files from a programming language into machine code for a given processor architecture.

2. The *linking*: this step produces the executable file by linking together the different modules produced at compilation.

3. The loading of the program and its preparation for execution: this step is handled by different tools from the operating system (program loader and *dynamic linker*). The goal is to load the program code into memory, to link it to the necessary shared libraries and to transfer the control to allow the execution of the program. These initialization steps are, therefore, performed at each execution.

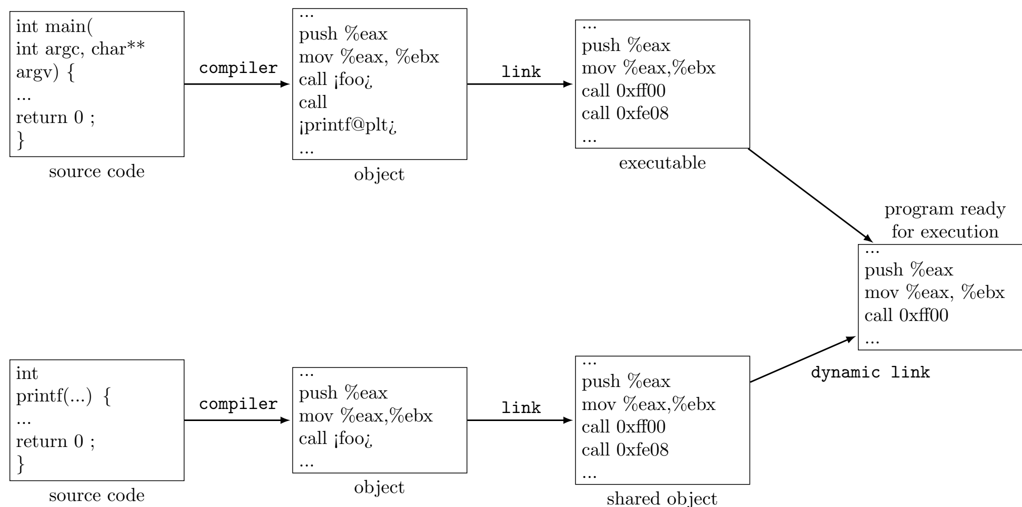


Figure 2.5: Compilation and execution cycle of a program

The first file type specified by the ELF format corresponds to the compilation step and is called object file. It cannot be directly used as an executable program, or by another program to extend its functionalities at execution. It is actually a container of data and code produced by the compilation of a source file, and can be associated with the rest of the program code at the *linking* step. In order to be shared between several compilation tasks, this file should be independent of any memory address that could be the cause of conflicts with other objects. Thus, the compiler creates a relocatable file. A relocatable file is composed of code independent of memory positions (PIC: *Position Independent Code*). In situations where the code requires hardcoded

addresses, for example, to reference data, the compiler uses symbols instead of addresses, which are later replaced by definite values during the *linking* phase.

The second type of file is the executable format. It is produced after the *linking* step. This format essentially contains all the information neccessary for execution. In particular, it may include references to shared libaries needed for running. The most commonly used library is LibC as it provides a large number of basic functions, used almost systematically in program.

The mechanism of library sharing has several advantages. It allows to share the development efforts as the functions developed in a library can be used directly from an executable at runtime. It also permits to benefit from these functions' updates without recompiling the program. Historically, this mechanism was proposed to limit the memory footprint of programs and to avoid to store the same code multiple times.

The third type of files, shared libraries or *shared objects*, is produced after the *linking* step too. However, they are not directly executable and can only be used when loaded by another program.

As the creation processes of the executable program does not consider information about the size and the memory position of the different shared libraries, the compiler has to establish indirection mechanisms. These mechanisms rely on tables whose fields are updated at *dynamic linking*. Concretely, to allow a program to call a function from a shared library, the compiler produces two elements:

1. A procedure (short and independent portion of code performing a specific task) within a table of functions called Procedure Linkage Table (PLT).

2. An uninitilized entry of address pointer in a table called Global Offset Table (GOT).

The GOT is an array of pointers, in which each entry of this table reference a function of shared library. The PLT is an array of small code the purpose of which is to fill the GOT with the correct pointer during the execution of the program. At the first call of the shared function the code executes the code contained in the PLT of the corresponding function. This code will resolve the address of the shared function and store it inside the corresponding GOT entry, and finally calls the shared function. On the next

call of the shared function the GOT is already filled so the PLT only call the shared function without the need to resolve it.

From this description, one can deduce that being able to read the GOT may allow an attacker to localize libraries in memory. Similarly, being able to write in this table may lead directly to taking control over the execution flow of the program.

2.1.3 Vulnerable points

Memory management as described in the previous section, is only a vision organized to make the code more understandable and its maintenance easier. But the processor has no knowledge about the semantics of the breakdown into sections, which is defined by the developer. In real terms, the processor do not make any difference between memory areas and the different ways to access them. This generic vision of memory serves to create memory corruptions, which are the causes of vulnerabilities.

In this section, it will be explained how programming errors can lead to memory corruptions. First, the two main types of memory corruptions will be presented, namely spatial errors and temporal errors.

Spatial errors

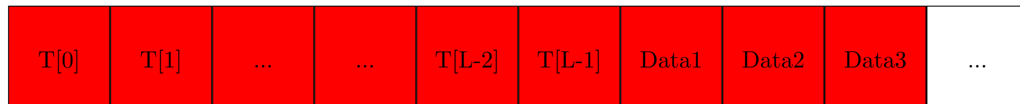
Spatial errors are some of the most common errors regarding memory management, and are often under the form of *buffer overflows*. Those happen when the index used to access a memory buffer is not correctly computed and addresses areas out of the buffer limits. This error may also occur when the value used by the program to index the buffer is directly linked to some user data without verification. Two types of errors can be distinguished: when the value is used for reading access and when it is used for writing access.

Buffer over-read If a spatial error occurs during a buffer reading, it can lead to data leakage. This kind of error is used to make information leak about the program. The "Heart Bleed" [19] attack is an example of these errors in which the attacker can recover sensitive data. In this attack, the number of bytes to read in a buffer was provided by the user request and used by the program without verification. With this knowledge, the attacker could recover data from other users, and even cryptographic keys from a server. This kind of attack is called *memory disclosure* or *memory leakage*.

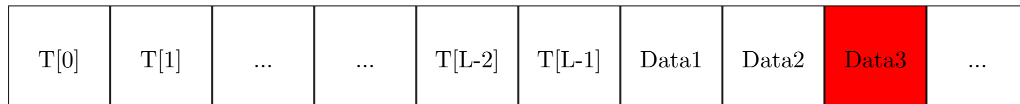
Buffer over-flow The same type of error may occur for reading accesses, creating what is called a *buffer overflow*. Depending on the memory buffer location, it can give an attacker the opportunity to modify the values of other variables (see Section 2.1.4). The most famous *buffer overflow* attacks are those located on the stack and which modify the code pointer of the function return address (see Section 2.1.4). *Buffer overflows* can also happen in other memory zones. This attack can be divided into two categories: linear or indexed *buffer overflows*. Figure 2.6 shows these two categories, with the attacker aiming at modifying the value of `data3` and red zones representing the modified memory cells.

1. linear *buffer overflow*: the writing continues beyond the buffer size and modifies all the data between the buffer and `data3`.

2. indexed *buffer overflow*: the attacker is able to modify the value of `data3` without compromising other data.



1. Linear *Buffer overflow*



2. Indexed *Buffer overflow*

Figure 2.6: 2 ways of realizing *buffer overflows*

Temporal errors

Another type of vulnerability can be exploited when the program reuses a pointer which is no longer valid. This generally occurs if there is a mismatch in the code parts responsible for allocation and deallocation of memory, and allows the attacker to write a memory word to an arbitrary address [9] [59] or to modify the called function by C++ object virtual methods [63]. Like

spatial errors, this kind of vulnerability may be used to read or write in unauthorized memory spaces, which can lead to the deflection of the program execution if this error is used to modify a code pointer (like *vtables*).

2.1.4 Vulnerability exploitation

In this section, we will show how the vulnerable points seen in section 2.1.3 can be used by an attacker to hijack the execution flow of a vulnerable program.

As presented thereafter, three main attack families can be distinguished:

- Attacks aiming only at the program data modification, meaning that the attacker uses the flaw to change the program behaviour without modifying its execution flow.

- Attacks by code injection, meaning that malicious instructions are placed into memory by the attacker and are executed with the help of the techniques described later.

- Attacks by code reuse, meaning that instructions already in memory are reused in a specific order and manner to perform malicious actions.

As a first step, we will focus on how to call the malicious code to execute rather than on its provenance. The techniques used to create payloads will be presented in section 2.1.5.

Data only attack

The first way to exploit a vulnerability is to use it to modify a program data in order to change the program behaviour. These attacks are called *data only attack* and are relatively simple, while difficult to counter.

Overwrite data A *buffer overflow* may be used to modify data next to the targeted buffer which will modify the execution flow of a program. In the following example, there is no verification on the size of the data given as argument to the `check_authentication()`, which will create a *buffer overflow* when calling the `strcpy` function.

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

int check_authentication(char* password)
{
        int auth_flag = 0;
        char password_buffer[4];

        strcpy(password_buffer, password);

        if(strcmp(password_buffer, "cafe") == 0)
                auth_flag = 1;

        return auth_flag;
}

int main(int argc, char* argv[])
{
        if(check_authentication(argv[1]))
                printf(" Access Granted\n");
        else
                printf(" Access Denied\n");
}
```

As previously explained, local variables of the function are stored on the stack. Since `auth_flag` and `password_buffer` are declared as local variables of the function `check_authentication`, they are next to each other in memory. If the attacker specifies a character string longer that 4 characters, he will be able to modify the value of the variable `auth_flag` and to force the return value of the function `check_authentication`. He will therefore obtain the access authorization without knowledge of the password.

Format String Exploitation In [58], Payer and Gross show how the control of the `printf` first argument provides access to write to any memory address. More specifically, the attacker controls the *tokens* used by the `printf` function (`%s, %p, %n, etc.`).

This kind of exploit is due to an incorrect utilization of the function

`printf()` by developers. The following example shows how the first argument of `printf()` can be controlled by the user.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  void main(int argc, char* argv[])
5  {
6          printf("Hello, ");
7          printf(argv[1]);
8          printf("\n");
9          // Instead of simply
10         // printf("Hello, %s\n",argv[1]);
11 }
```

If the attacker uses tokens such as `%s,%p,%n` in the first argument of the program (`argv[1]`), the tokens will be interpreted by `printf`. By correctly formatting this character string, the attacker can control both the writing address and the value to write. `%n` allows to print the number of characters written by `printf` inside a pointer. By carefully crafting the string argument passed to `printf` the attacker can write any value anywhere in memory. For more practical information on how this attack works please refer to [58].

Stack overflow

The first *Stack overflow* exploit dates back from 1960s. It consists in filling a memory buffer located on the stack with more data than it can contain, in order to overwrite or modify critical data. The main objective of these attacks is to compromise the return address of the current function so that the execution flow of the program is redirected to the attacker payload when the function returns.

Aleph One [54] presents in details this vulnerability and how to exploit it. As explained previously, at a function call, the following items are sequentially pushed on the stack.

- the arguments of the function call.
- the return address of the function.
- the address of the previous local frame.

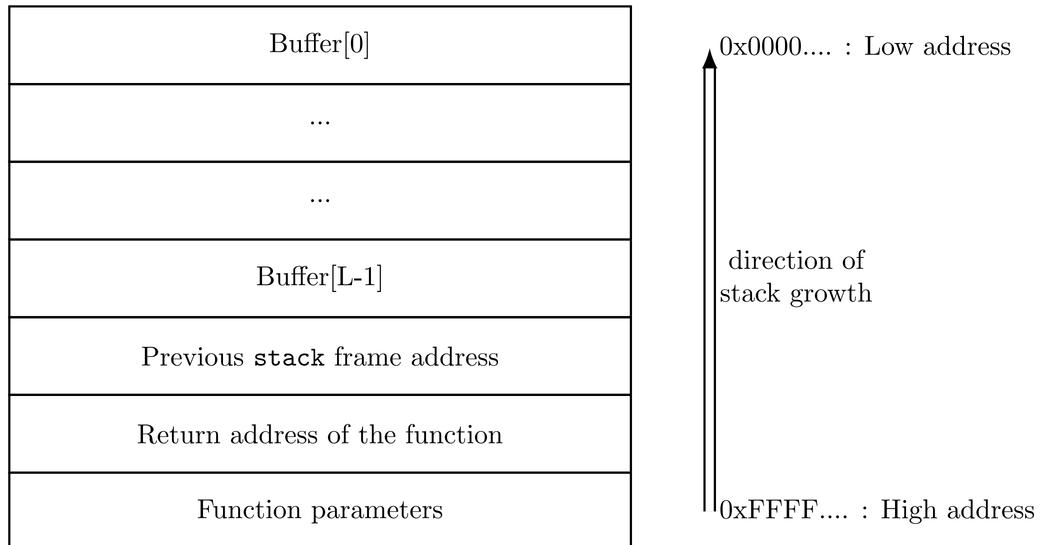


Figure 2.7: Overflow in the stack

- the local buffers and the local variables.

The surplus of items copied in the stack should overwrite the return address of the calling function. Thus, after the function call, the address of the next instruction to execute can be chosen by the attacker in order to execute malicious instructions.

Figure 2.7 shows a stack after the function call. The stack contains a local buffer of L elements. The objective of the stack overflow attack is to fill this local buffer with L+2 elements. The Lth and the $L+1^{th}$ elements will overwrite the address of the previous local frame and the return address of the function, respectively. At the end of the function execution, the value of the $L+1^{th}$ element will be put in the `%eip` register (instruction pointer). By copying the start address of a malicious code into the $L+1^{th}$ element, it is possible to hijack the execution of the program.

Following is a simple example of redirection:

```
1 void hackExample(){
2         printf("\n****** Redirection of the 
             function********\n\n");
3 }
4
5 void save_name(char *input_name) {
```

```
        char buffer[2];
        strcpy(buffer, input_name);

        printf("Your name, %s, was successfully saved\n", buffer);
}

int main(int argc, char *argv[]) {
        if (argc != 2) {
                printf("Usage : %s <Your name>\n", argv[0]);
                exit(0);
        }

        save_name(argv[1]);
        printf("Process normaly executed\n");

        return 0;
}
```

```
$ ./test $(python -c 'print "aaaaaaaaaaaaaa" + "\xc4" + "\x84" + "\x04" + "\x08" ')
Your name, aaaaaaaaaaaaaat?, was successfully saved

******Redirection of the function********

Segmentation fault (core dumped)
```

With the help of gdb and the `disassemble` command, it is easy to retrieve the address of the function `hackExample : 0x080484c4`. By creating an *overflow* and injecting the address in the arguments' characters, it is possible to execute the `hackExample` function without calling it in the source code.

Currently, numerous solutions exist to counter this type of flow control exploitation, such as the ASLR countermeasure or canaries. To successfully run the example above, it is necessary to disable some protections:

- ALSR: # echo 0 < /proc/sys/kernel/randomize_va_space;
- the stack protector, at compilation: -fno-stack-protector.

Heap overflow

Unlike the stack, the heap does not contain the return addresses of functions. It is however possible to write a chosen integer almost everywhere in memory. To achieve this, several attacks have been proposed, as well as associated countermeasures which will be presented in later sections. The basic techniques for control flow hijacking through *heap* corruption, presented in [42] and in [9], exploits the `unlink()` and `frontlink()` macros, respectively used for allocation and deallocation of *chunks*.

To understand the attacks based on *heap* overflow, the mechanics of the functions `unlink()` and `frontlink()` will be first explained.

Let us consider the example of a *bin* composed of three free *chunks* as presented on Figure 2.8.

Each free *chunk* is characterised by two main fields: the address of the previous *chunk* named `bk` (*backward*), and the address of the next *chunk*,

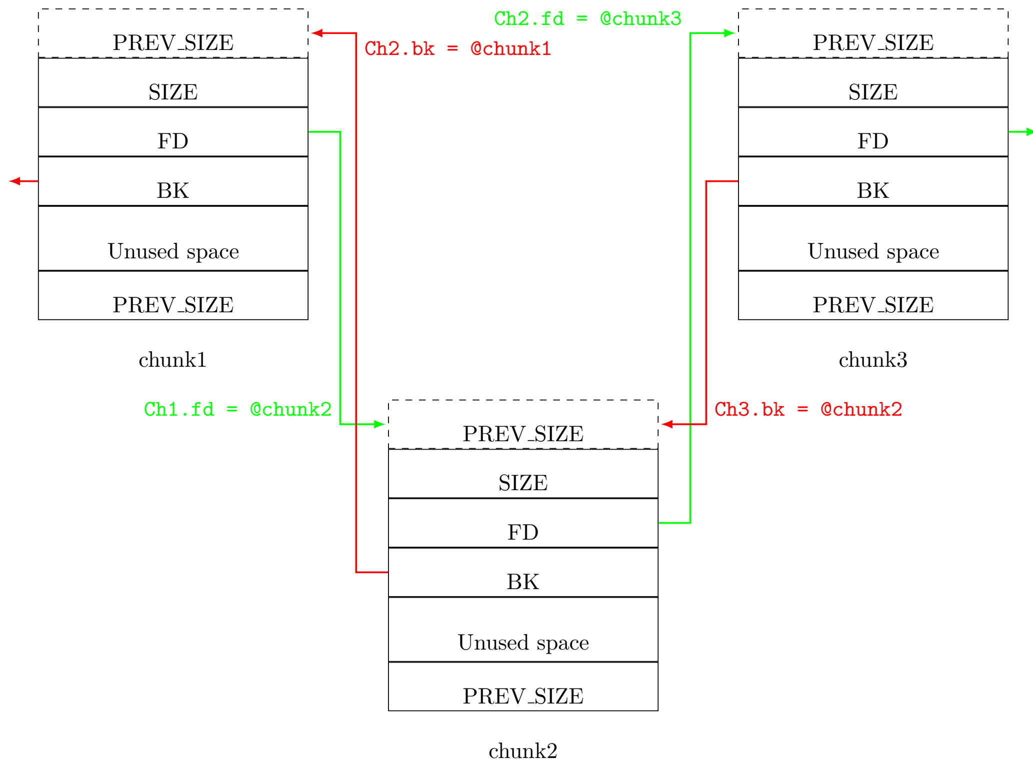


Figure 2.8: Example of a *bin*

named `fd` (*forward*). For example, in the case of *chunk* 2, `ch2.fd` contains the address of the *chunk* 3 and `ch2.bk` contains the address of the *chunk* 1. When a free *chunk* is allocated (for example, *chunk* 2), it is deleted from the *bin*. The *bin* should reshape its doubly linked list to, exclusively, contain the *chunks* 1 and 3. This action is performed by the macro `unlink()`, the code for which is presented below:

```
1 #define unlink( P, BK, FD ) {
2             BK = P->bk;
3             FD = P->fd;
4             FD->bk = BK;
5             BK->fd = FD;
6 }
```

In our example, outlined on Figure 2.9, the macro performs the following operations:

1. Copy the value of `ch2.bk` at address (`ch2.fd`) + 12 which contains the address of `ch3.bk`.

2. Copy the value of `ch2.fd` at address (`ch2.bk`) + 8 which contains the address of `ch1.fd`.

If an attacker modifies the header of the *chunk* 2 (mainly the content of `ch2.bk` and `ch2.fd`), then he can write any integer value at an arbitrary address. In our example, he can write the value of `ch2.bk` at address (`ch2.fd`) + 12.

A simple example of exploitation of this vulnerability is presented in [42]. It is about creating an *overflow* in a buffer A, in the heap. With wisely chosen values, the *overflow* creates a fake free *chunk* B, adjacent to A. Given the characteristics of B, the macro `unlink` will be called during the deallocation of A and will modify the entry of the function `free()` in the `GOT`, which will redirect towards malicious code. At the next call to `free`, the malicious code will be executed.

Another possible way to highjack the execution flow is to exploit the `frontlink()` macro. It is used to replace a freed *chunk* in a *bin*. It is stated in [42] that taking control of the execution flow through `frontlink()` is less flexible and more difficult than with `unlink()`.

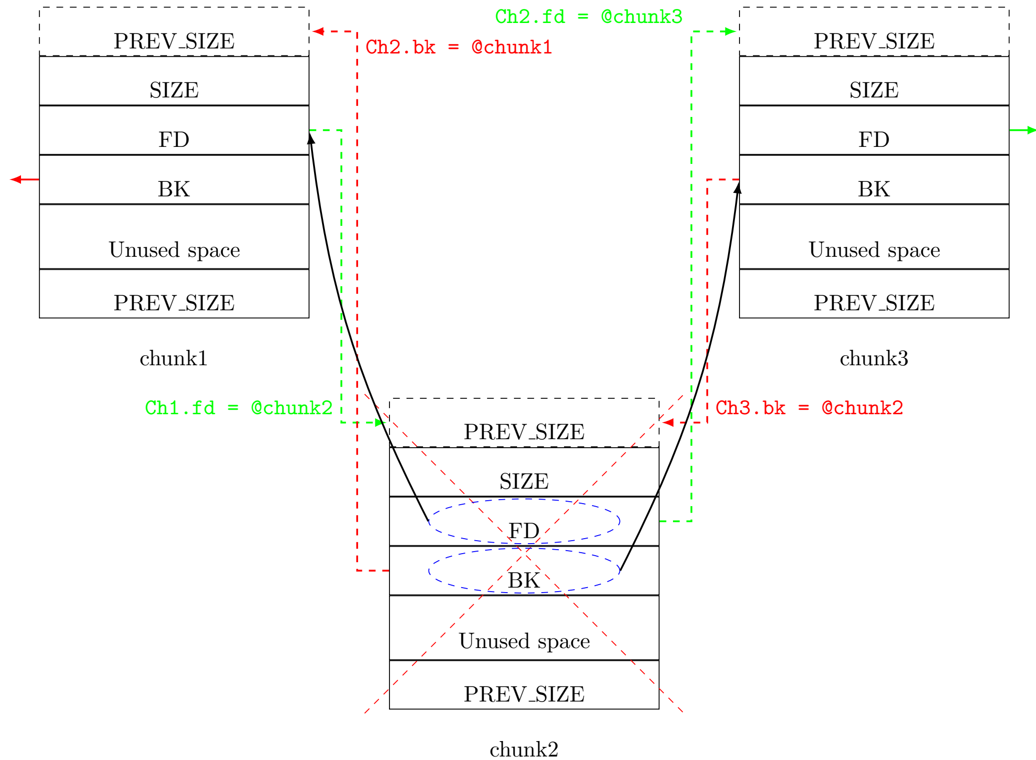


Figure 2.9: *chunk* modification in the *heap*

This section only focuses on the basic exploitation of the heap to hijack to execution flow. More elaborated solutions exist (and they are therefore more difficult to use), presented in several publications such as [59] and [41], where exploits on the memory allocator `Malloc` are described.

Data and bss overflow

Like the stack or the heap, the `.data` and `.bss` segments (Figure 2.10) can be exploited to corrupt data, and also to deflect the execution flow by modifying, for example, important variables such as function pointers.

In the segment `.bss`, the flow diversion can be obtained with an *overflow* of a *buffer* located on top of a function pointer. By copying a L+1th element, the function pointer can be overwritten. It may then point to a malicious function.

2.1.5 Payload creation method

The previous section presented solutions to divert a program execution, in other words, to make a program execute unexpected malicious code at a specific moment. This section addresses the provenance of the malicious code, which is:

- either injected: the attacker writes his own code and deflects the flow of the target program to execute it.
- either reused: the attacker uses code parts of existing programs, executed at specific moments and in a specific order.

Code injection

The attacker may leverage on user inputs to save data into memory, that will be interpreted as code by the processor.

Standard code injection: the shellcode A *shellcode* is simply a character string which can be interpreted as executable code. Historically, in [54], once interpreted, this character string, once interpreted, launched a command interpreter called *shell*. The advantage is that the *shell* has the

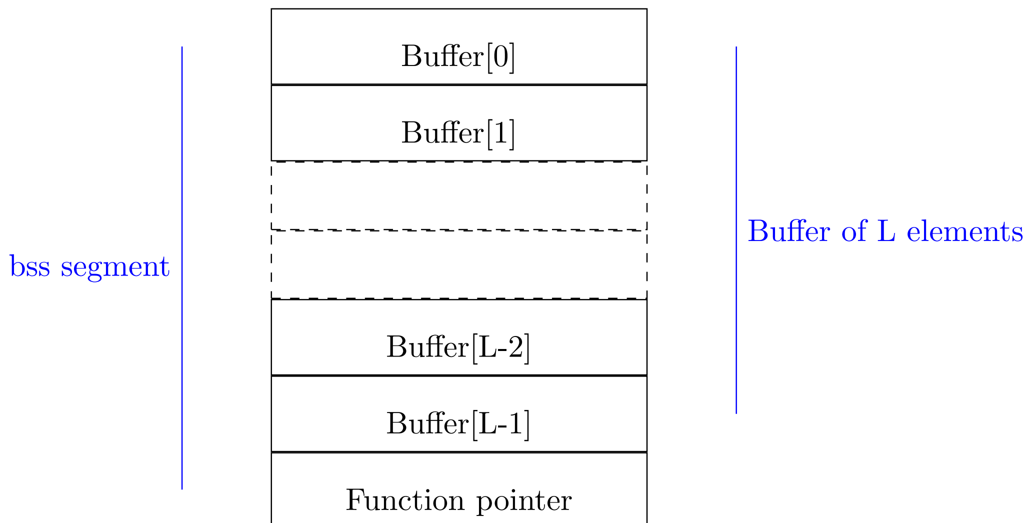


Figure 2.10: `bss` segment

same privileges as the vulnerable program. Thus, if the vulnerable program is SUID root, the shell commands will have the same privileges as the root user.

Generic *shellcodes* have been programmed for different architectures, such as the ones proposed in [62].

To create an operational *shellcode*, it is necessary to respect some rules:

1. It can be written directly in assembly or in C. In that case, it will be necessary to compile it with the option -static so that it will contain the code of the called external libraries. The produced binary code will then be transformed into a character string.

2. It must be of small size in order to be exploitable on small size buffers.

3. If the *shellcode* is injected under the form of *C-String*, it must not contain the byte NULL which would prematurely interrupt the *shellcode*.

4. The *shellcode* should not contain absolute addresses.

The *shellcode* proposed in [54] and shown below opens a *shell* by executing the command "/bin/sh" via a system call.

```
1 char shellcode[] =
2 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46"
3 "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
4 "\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

A simple case of utilization of this shellcode is to overflow the return address of a function in the stack with the start address of the shellcode instead.

Code injection in a JIT compiler Just-In-Time compilers (JIT) are programs which, while running, quickly compiles source code and executes it. The goal is to achieve performances similar to those of compiled languages while preserving the flexibility of interpreted languages. There are JIT compilers in the .NET framework, in Java virtual machine and in Web browsers.

JIT compilers are critical regarding security: a website might convey code which, once compiled, behave maliciously. This is why code compiled at runtime is always compiled in a low risk environment, also called *sandbox*,

where system calls are limited. JIT compilers generally do not compile the entire code, but only the most used parts (the compilation should be fast in order to avoid slowing down the launching/initialization).

An attacker can inject arbitrary code by using the following technique: declaring a variable as the result of successive XORs between different 32-bits integers. This code can then be compiled at runtime to speed up its execution (especially if it is used in a loop).

```
1 var a = (0x11223344^0x44332211^0x44332211^...)
```

Let us dissassemble the produced code:

```
1   0:  b8 44 33 22 11        mov $0x11223344,%eax     mov eax,0x11223344
2   5:  35 11 22 33 44        xor $0x44332211,%eax     xor eax,0x44332211
3   a:  35 11 22 33 44        xor $0x44332211,%eax     xor eax,0x44332211
```

This code has the expected behaviour: it computes the initial value of `a`. However, let us see what would be the effect of executing this code from the address 1 instead of address 0.

```
1   1:  44                    inc %esp                  inc esp
2   2:  33 22                 xor (%edx),%esp           xor esp,DWORD PTR [edx]
3   4:  11 35 11 22 33 44     adc %esi,0x44332211       adc DWORD PTR ds:0x44332211,esi
4   a:  35 11 22 33 44        xor $0x44332211,%eax      xor eax,0x44332211
```

This code behaves differently, as the processor decodes it with an offset of one byte (the first byte b8 is ignored). This is possible for processors with instruction sets of variable size (eg. x86) where instructions are not aligned on 32 or 64-bits multiples. In reality, it can be shown that any program can be injected and made executable in this way. Then, it remains to find a vulnerability to deflect the execution flow of a program to this shellcode.

Code Reuse Attack

Payload creation via code injection is one of the earliest techniques, and associated countermeasures such as Address Space Layout Randomization (ASLR) or Data Execution Prevention (DEP) have already been deployed. In order to bypass these protections, attackers can choose the code reuse technique. The principle is to modify the execution flow of the program in order to execute a sequence of existing instructions in a specific order.

Return-to-libc The first code reuse attack to appear was presented by Solar Designer in "lpr LIBC RETURN exploit" [28], later renamed "Return-to-libc". In this, the attacker uses an exploit to store the parameters of a function from the shared library `libc` in the stack data, and modifies the return address with the address of the desired function. Generally, the attacker tries to call the `system(...)` function which simply executes the *shell* command given in argument.

Return-oriented programming (ROP) The concept of ROP was proposed first by Hovav Shacham in 2007 [64]. This method is the generalization of the attack "Return-to-libc". It is the method of choice for payload creation (along with JOP).

It consists in using short sequences of code available in the binary or in the libraries linked to the application which, when sequentially executed, conduct the operation desired by the attacker. These short code sequences, called gadgets, must end with the instruction "ret" or "0xc3" in hexadecimal, for x86 assembly.

To execute malicious operations, it is first necessary to find available gadgets. This step unfolds as follows:

1. First find bytes equal to 0xc3 (instruction "ret") in the binary code.
2. For each "ret" byte found, check if the previous instructions are valid. In reality, only the preceding 20 bytes are verified; that is an arbitrary limit of the searched gadgets size.
3. Establish a list of gadgets with their addresses and functionalities.

Some tools are available to help in gadget researching, such as in [39] or [48].

Gadgets offer plenty of basic functionnalities:

- loading a constant value into a register.
- loading a constant value into memory.
- saving a register value into memory.
- arithmetic operations.
- function calls.

- conditional or unconditional branchings

Gadget exploitation and sequencing is possible because of the "ret" instruction. During the program execution in assembly, when this instruction is met, the top value is popped from the stack and put into the `%eip` register, containing the address of the next instruction to execute.

Because a gadget is composed of one or several instructions followed by the instruction `ret`, deflecting the execution flow can be done by simply using an overflow to create a chain of gadget addresses, interspersed with constants if needed.

Let us consider a program calling a function `fct1` with two parameters and containing a local buffer of size L. The stack content after the function call is presented on the left of Figure 2.11. For example, suppose a ROP attack whose goal is to add two constants `cst1` and `cst2`, given by the attacker, and to save the result in a register. For that, we assume that three gadgets are available:

- `pop %eax ; ret ;`: This gadget pops the top value from the stack and saves it in the register %eax. Consider that the address of this gadget's first instruction is @gad1.

- `pop %ebx ; ret ;`: This gadget pops the top value from the stack and saves it in the register %ebx. Consider that the address of this gadget's first instruction is @gad2.

- `add %eax, %ebx ; ret ;`: This gadget adds the content of registers `%ebx` and `%eax` and saves the result in `%eax`. Consider that the address of this gadget's first instruction is @gad3.

The left part of the Figure 2.11 presents the stack after the call of the function `fct1`, while the right part shows the stack after the *overflow*.

Figure 2.12 presents the state of the stack and of the register `%eip` for each instruction:

- t : the instruction `ret` of the function `fct1` is executed.

- t + 1 : the instruction `pop %eax` of the gadget 1 is executed, %eax=`cst1`.

- t + 2 : the instruction `ret` of the gadget 1 is executed.

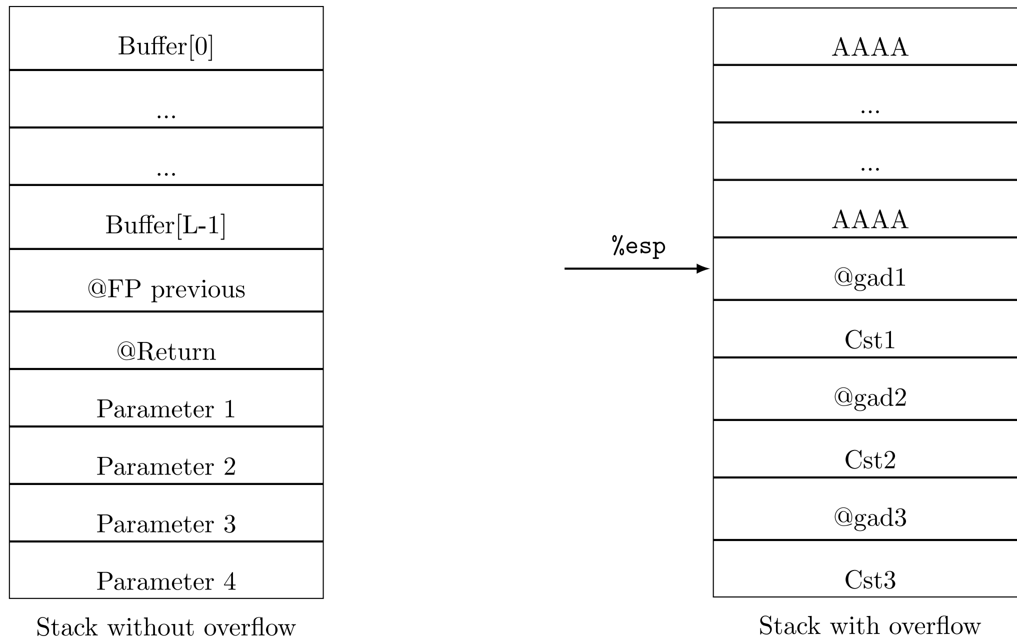


Figure 2.11: Stack before and after a ROP attack

- t + 3 : the instruction `pop %ebx` of the gadget 2 is executed, %ebx=`cst2`.
- t + 4 : the instruction `ret` of the gadget 2 is executed.
- t + 5 : the instruction `add %eax, %ebx` of the gadget 3 is executed, %eax=`cst1`+`cst2`.
- t + 6 : . . .

Thus, at the end of the execution, register `%eax` contains the value computed by `cst1`+`cst2`. The same idea, applied to different gadgets, allows the attacker to realize any operation.

Jump-oriented programming (JOP) Another attack based on code reuse was proposed in "Jump-oriented programming: a new class of code-reuse attack" [14] in 2011. Named Jump Oriented Programming (JOP), it relies on reuse of code ending with the `jump` instruction (and not by `ret` as in ROP attaks). These instruction blocks are also called gadgets.

The first step consists in searching valid gadgets. For that, an algorithm looks into the execution segment for the character string 0xFF, which is the bytecode for the instruction `jump`. The exploitation based on these gadgets is different from the ROP attack. For JOP attacks, the addresses of the gadgets should be placed in memory in an *initializer table* via a set of instructions named *initializer gadget*. Then, the control of the flow is handled by a special gadget, the *dispatcher*, which sequentially executes the different gadgets.

Figure 2.13 shows the mechanism of a JOP attack with the order of the executed `jump` instructions in red color.

It works as follows:

- The dispatcher obtains (or computes) the address of the next gadget to execute, stored in the dispatcher table.
- (1) The dispatcher executes a jump at the address previously obtained.

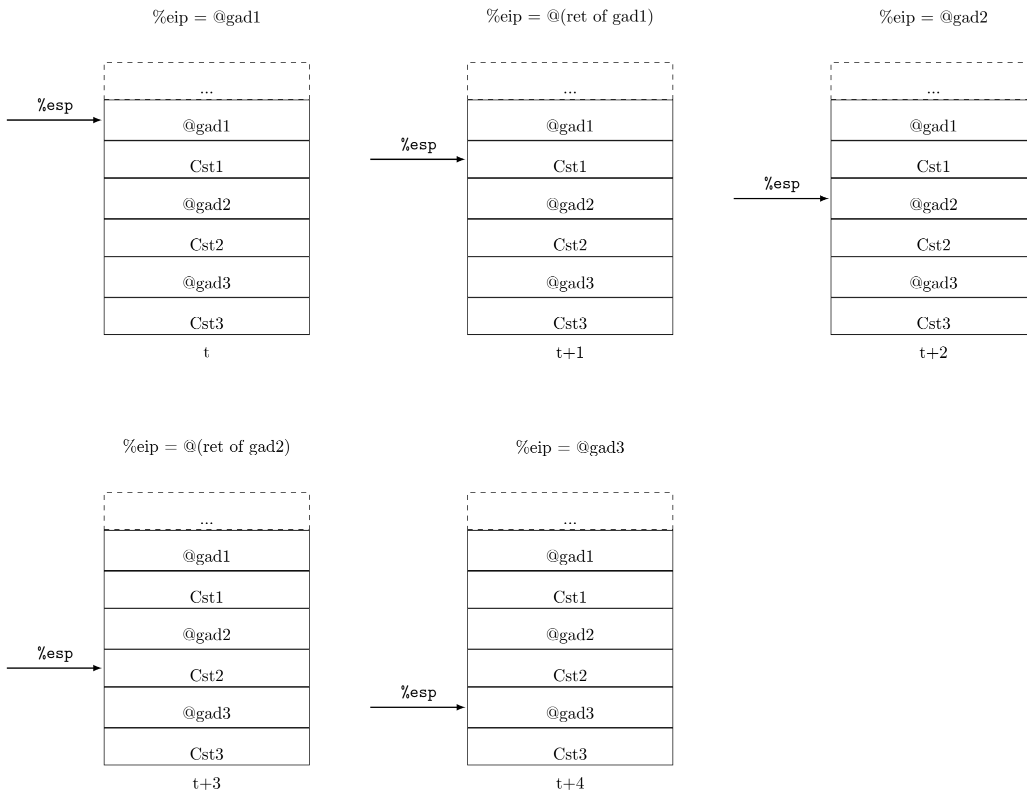


Figure 2.12: Stack during a ROP attack

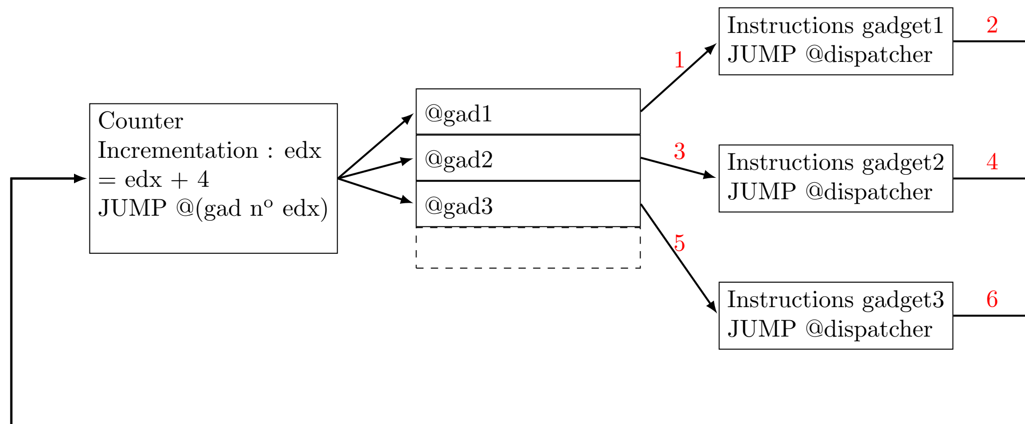


Figure 2.13: JOP example

- The instructions of the gadget are executed.
- (2) At the end of the gadget, a jump instruction to the dispatcher is executed.
- The dispatcher obtains (or computes) the address of the new gadget and executes a jump
- (3) and so on.

Call Oriented Programming Similarly to the JOP exploit, COP techniques used gadgets ending with an instruction modifying the execution flow of the program. However, these gadgets ends with an indirect `call` instruction. COP is very similar to JOP, but the use of `call` modifies the program stack. For example, on x86, it will automatically save the register `%eip` on the stack. The behavior of `call` can differ depending on the architecture. These particularities can be used by an attacker to build more complex payloads.

Counterfeit Object-Oriented Programming Schuster et al. present in the "Counterfeit Object-oriented Programming" [63], a generic method to exploit C++ objects' `vtables`. One of the particularities of object-oriented languages is the use of `vtables` for object virtual functions. The method shows how a counterfeited object can be used by an attacker to use the `vtable` of this object as a ROP attack *dispatcher*.

Signal Oriented Programming Presented by Erik Bosman & Herbert Bos in [16], this special variant of ROP uses a *buffer overflow* in the signal manager (*signal handler*). At the reception of a signal, the kernel saves the execution context of the program in a structure called `uc_mcontext` and pushes it on the stack. It then gives the control to the *signal handler*. After executing the signal manager, the kernel returns the control to the program by using the previously saved structure `uc_mcontext` to restore the context of the program. By creating a *buffer overflow* on the stack, it is possible to modify the `uc_mcontext` structure to modify the instruction pointer and the different flags stored there. Thus, at the return of the *signal handler*, the modifications of the `uc_mcontext` structure makes any system call (*syscall*) possible.

2.1.6 Synthesis

In the previous section, we introduced the architecture of a program in memory, and the low level programs' intrinsic vulnerabilities. These vulnerabilities primarily exist because the processor does not consider the segmented representation of memory introduced by the programmer in the ELF format. The modus operandi of the attacker is always to exploit this generic aspect of the processor, to access or modify memory areas that are security critical due to their semantics "Eternal War in Memory" [69] proposes the representation of the attack path of different exploits, as presented in Figure 2.14.

The graph in the figure well summarizes the possible attacks. It is important to note that all the attacks starts with a spatial error (*Make a pointer go out of bound*) or by a temporal error (*Make a pointer become dangling*). The green sets show the different protection categories, protecting against the different phases of the exploit.

MS : Memory Safety

DI : Data Integrity

- **CI** : Code Integrity
- **CPI** : Code Pointer Integrity

DSR : Data Space Randomization

- **ASR** : Address Space Randomization

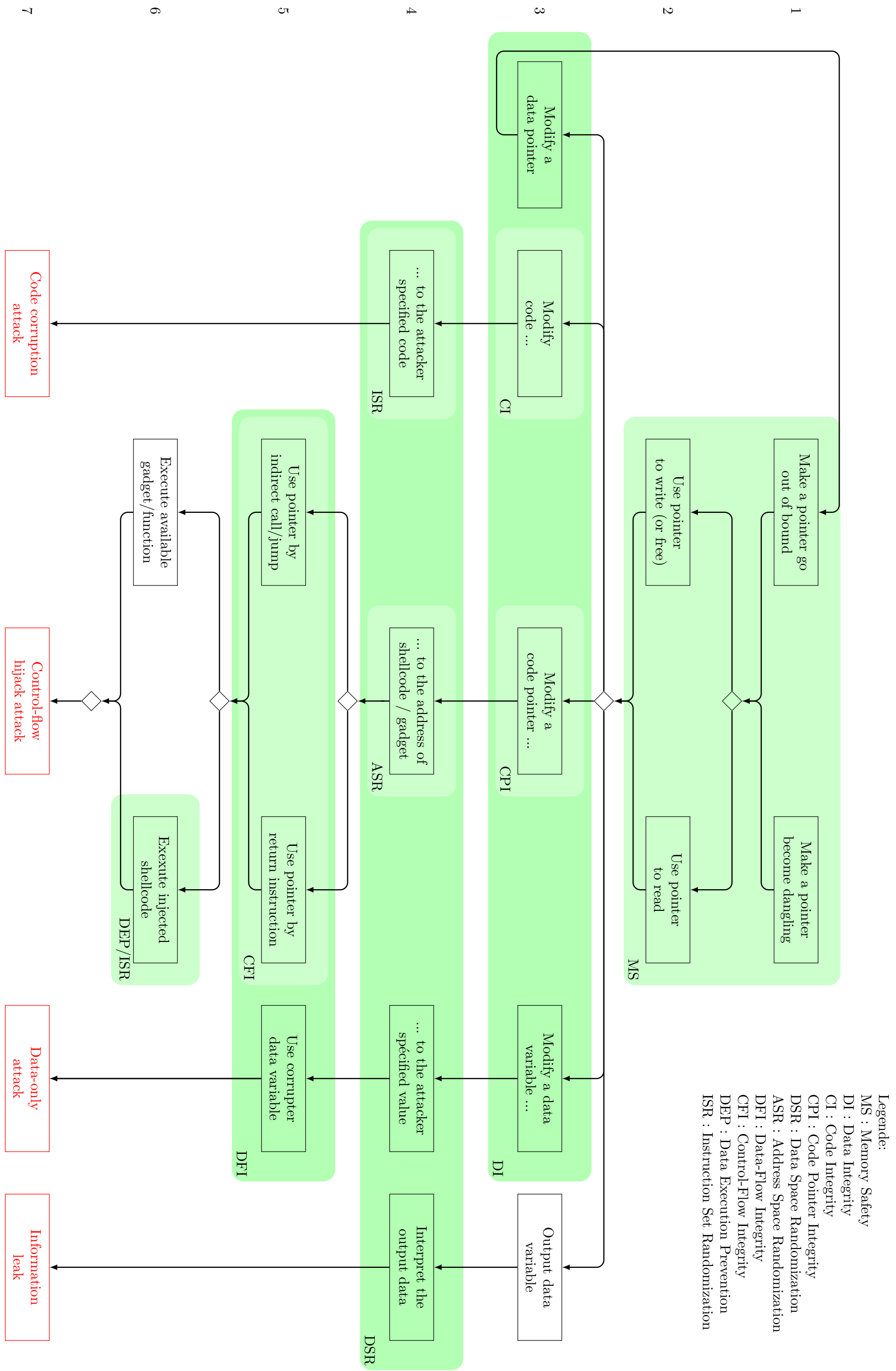


Figure 2.14: The different attack models

DFI : Data-Flow Integrity

CFI : Control-Flow Integrity

DEP : Data Execution Prevention

ISR : Instruction Set Randomization

2.2 Hardware countermeasure

In this section we present countermeasures based on hardware implementations. Some of them are transparent for software operation while others need software modifications in order to operate properly. The countermeasures commonly deployed on recent architectures as well as academic research on the subject will be described. The impact on performance of the countermeasures is also indicated as and when available.

2.2.1 Existing Hardware Protections

In this section we will first present hardware countermeasures currently deployed on recent architectures.

Data Execution Prevention

The countermeasure *Data Execution Prevention* (DEP), which is also called W$\oplus$X, (W XOR X), is a protection mechanism intended to make memory pages exclusively executable, or writeable. Code pages are, therefore, executable but accessible for reading only. On the contrary, the heap or stack can be accessed for writing but cannot contain executable code. It is an effective protection against code injection, however it does not protect against ROP and related attacks nor against hardware attacks such as fault injection.

Thus, an attacker who controls the contents of a memory zone in write mode may not use it to inject code. The first (software) implementation of this protection was realized under the name of PaX protection. It consisted in overloading the use of *supervisor* bit in order to simulate the NX bit. Another method consisted in separating the writable data and executable code in two

distinct memory zones. Today, this countermeasure is directly supported by the processor. Depending on implementations, software or hardware, several names have been given to this countermeasure:

- NX bit: generic term;
- Wˆ X: OpenBSD software implementation;
- Exec Shield: Red Hat software implementation with emulation for Intel x86;
- PAX: software implementation on Linux;
- XD bit: hardware support for Intel processors (x86_64);
- Enhanced Virus Protection: hardware support for AMD processors;
- XN : hardware support for ARM processors.

The hardware implementation is simple and corresponds to a single bit, in the table of pages, showing for each page if it can be accessed in executable or write mode. The protection via NX bit is disabled for some old software which may need to see their own code modified. JIT compilers can use this protection by making the memory pages containing code compiled on the fly non-writable afterwards. However this leaves a time window during which the executed code could be accessed in write mode. Nowadays this hardware countermeasure is widely deployed and its impact on software performance is considered to be null.

Control-flow Enforcement Technology

In *Control-flow Enforcement Technology Preview* (CET) [36], Intel presents its countermeasures to reinforce execution flow control. The two proposed protections are a *Shadow Stack* and the addition of instructions to check indirect jumps. While the shadow stack is a powerful solution for inter-procedural CFI, it does not provide any other guarantees. On top, Intel introduces an interrupt code called *Control Protection Exception* (`#CP`), which is the exception used by these new countermeasures when a security bypass is detected.

Shadow Stack Intel proposes the setup of a *Shadow Stack* compatible with the modes `USER` and `SUPERVISER`[1]. To achieve this, the behaviour of instructions `CALL` and `RET` is modified. Instruction `CALL`, on top of storing the return address in the software stack, will store a copy of this return address in a *Shadow Stack* placed in another memory page. In order to do that, a new register called `SSP`, which contains the address of the *Shadow Stack* top, is added to the processor. The instruction `RET` is also modified to fetch the two return addresses and to compare them. If the two addresses are different then the processor raises exception `#CP`.

The memory page controller ensures the protection of the *Shadow Stack* memory zone. To this effect, a marker is added in the controller, enabling to mark a page as used by a *Shadow Stack*. This allows to prevent write/read of the protected stack by the protected software. Since the protection of the memory zone of the *Shadow Stack* is undertaken by the memory page controller, it is not available in 8086 compatible mode since the page controller is not activated in this mode.

There is no communication about the performance of this countermeasure, but one may think its impact is low, since it only adds one additional memory write to the instruction `CALL`, and one additonal read when calling the instruction `RET`.

Indirect Branch Tracking The second countermeasure is the addition of an instruction `ENDBRANCH` (`ENDBR`), which is intended to validate indirect jumps when they are executed. To this end, instruction `ENDBR` is added after each indirect `CALL` and `JMP`. At execution, the processor expects to find the instruction `ENDBR` after jump instructions, and if this is not the case the exception `#CP` is raised.

This countermeasure seems not to prevent the modification of code pointers, but rather mitigates the issue of JOP gadgets proliferation in software which use instructions of varied sizes, which is the case with x86. On top, the mandatory presence of instruction `ENDBR` after a `CALL` must make the discovery of gadgets fulfilling this condition more difficult.

We did not find information quantifying the impact of this countermea-

[1] In this case, the protection of the page *Shadow Stack* is not anymore ensured

sure, whether in terms of performance or in terms of reduction of the number of available gadgets.

Memory Protection Extension

Memory Protection eXtension (MPX) [1] is an extension for Intel processors enabling the protection of memory access. It is integrated in the Skylake family (sixth generation of Intel microarchitectures). MPX is an extension of the x86 instructions set to support pointers check:

- 4 new 128 bit registers (BND0 to BND3);
- 4 new check instructions (`BNDMK`, `BNDCL`, `BNDCU`, `BNDCN`) ;
- 4 new data load/save instructions on pointers (`BNDMOV` (2), `BNDLDX`, `BNDSTX`) ;
- 1 new interrupt #BR.

On execution, the new MPX instruction set allows to verify memory addresses which may be accessed by a software. Thus, a process may not access a memory on which it has no rights and is also protected against attacks of the *buffer overflow* type. The principle is that each pointer is associated to an address range in which it can move (spatial security, not the temporal). If a pointer which is out of its allowed address range is de-referenced, an exception is raised.

Thus MPX consists in a set of new registers and instructions, which are accelerated by the CPU in order to minimize the additional cost of this verification. This should allow to implement a fast access to metadata associated to the pointers, which are stored in a *Shadow Memory*. The Intel MPX technology is now functional with the latest versions of Linux kernels such as those of Fedora and Ubuntu (kernel 4.1 is recommended and must have the *flag* CONFIG_X86_INTEL_MPX activated) and the latest versions of the *gcc* compiler ($\geq$ 5.2 recommended).

Intel MPX needs a recompilation of executables via GCC options. Readers are invited to refer to document [35] for more explanations. Code compiled with the MPX protection remains compatible with old executables

which will not benefit from the protection. At compilation, the code undergoes a static analysis to detect pointers which access the memory. The bounds of these pointers are calculated and the code is then modified to activate checks at runtime. The compiler is in charge of suppressing predictable checks during static analysis. The source code may need minor modifications because some ambiguities must be cleared in very specific cases. The check of pointers may also be deactivated when starting the execution in order to increase performance.

The pointer bounds calculated by the compiler are indicated in a *Bounds Table* where access is done via a 2-level tree structure (*tree*) (see Figure 2.15 page 71). This management is necessary because 4 BND registers are not enough to save all the data linked to all the software pointers. For a given pointer, there may also be different possible accesses and, thus, different bounds.

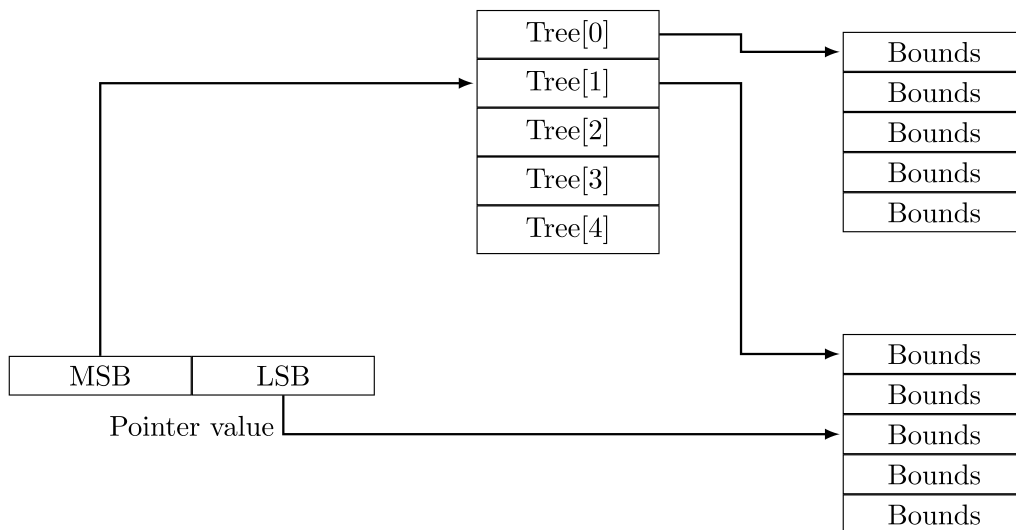


Figure 2.15: Example of *2*-level Lookup-Tree

In the case of 32-bit pointers, each Tree is made with 16 bits. One should note that the Tree and Bounds tables do not necessarily use memory space; the operating system only creates memory pages when they are accessed for the first time. Thus, it is not necessary to fill the Tree table with the maximum of 232 Bounds sub-tables.

However, it is important to note that a software compiled for MPX may use a lot of memory in the worst case (about 500% more). Moreover, each time a pointer bounds information are loaded, the processor must read data in a memory out of the software virtual address space, which impacts the bandwidth related to the Memory Management Unit (MMU) (translation of virtual addresses in physical addresses). The reader is invited to refer to the following study: [50] about performance with benchmarking in order to obtain more detailed information. A thorough evaluation of these aspects must then be performed on applications sensitive to performance, before deployment with Intel MPX technology.

Narrowing The case of structures is managed in a specific way. On the one hand, it is not possible to differentiate a pointer on a complete structure from a pointer on the first element of that structure. On the other hand, tables of structures present an issue, because the number of entries created in the Bounds Table should be equal to the number of elements in the table.

The different elements of a structure or a table are, thus, only partially protected against *overflows*.

For example, the following case is protected:

```
1 struct myStruct {
2         int a;
3         char buf[10];
4         int b;
5 }
6 myStruct s;
```

Here, `s.a` and `s.b` are protected against *underflows*/*overflows* on `s.buf`.

However, it is not the case in the following examples:

```
1 struct myStruct {
2         char buf[10];
3         int b;
4 }
```

```
5 myStruct s;
```

Here, one may not be able to distinguish a pointer on `buf` from a pointer on `s` (the compiler is able to make the distinction, but not the hardware architecture). Thus, `s.buf` may provoke an *overflow* on `s.b`. Similarly, if we look at the following case:

```
1 struct myStruct {
2         int a;
3         char buf[10];
4         int b;
5 }
6 myStruct s[10];
```

Here, `s[x].buf` may cause an overflow on the complete table `s`.

The *narrowing* method is described in [35]. Other technical information on MPX implementation can be found in the thesis of Nagarakatte [51] and in WatchdogLite [49], in an improved version. A development manual is also available on the Intel site [2].

Pointer Authentication

ARM Pointer Authentication, or ARM-PA, is a new security technology developped by ARM and integrated into their ARM v8.3 architecture (2016). It aims at protecting the value of some pointers while they reside in memory.

The specification is not publicly available, but an official change log is available on their blog. Detailed explanations are available in an official ARM presentation. Other useful resources are Qualcomm's whitepaperand Liljestr and et Al. analysis.

The most typical vulnerabilities found in a program are memory corrupting vulnerabilities such as buffer overflows. The most typical way to exploit such vulnerabilities is often to change the value of function pointers, or return adresses, to change the flow of execution. This allows an attacker to remotely execute some malicious code in the target program.

ARM-PA protects the value of such pointers while they reside in memory.

If an attacker wanted to use a vulnerability to replace a pointer's value, it would be impossible for her to craft a valid value.

Thus, ARM-PA prevents attacks that aim at altering the execution path using code pointers, such as ROP, JOP and others.

However ARM-PA has some limitations. Although it is possible to authenticate any pointer, including data pointers, ARM-PA does not prevent the pointed-to data to be altered. It means that it cannot be used to mitigate data-based attacks all by itself. However, it should be noted that in most cases it will effectively make it harder for an attacker to do a data-based attack.

Since ARM-PA protects pointers only while they reside in memory, it does not prevent physical attacks that could alter the value of a register, or the instruction being executed.

Since ARM-PA protects only pointers, it does not prevent attacks that alter the static part of the control flow. These are typically physical attacks that can skip or replay an instruction.

ARM-PA can only be as efficient as the key management and signing architecture is. If a vulnerability allows an attacker to access the key, to make the software sign a malicious pointer, or if an attacker can guess the key (e.g. due to weak key generations), then ARM-PA becomes useless.

In order to use this protection, a few requirements must be fulfilled:

- One needs to use a compatible processor: AArch64 on an ARM-V8.3A or greater.
- The program needs to be altered. If one wants to protect return address pointers, this can be done by a compatible compiler (GCC version 7 or greater). For other pointers, one needs to manually edit the code.

Some new instructions are encoded in the NOP space of previous architectures, meaning that the resulting binary is backward compatible with other ARM-V8 processors.

The way ARM-PA operates is described below.

ARM-PA is based on the fact that not all 64 bits of a pointer are used on the AArch64 architecture. Typically, on Linux, for a page size of 4KB and

three levels of page tables, only the 40 least significant bits (LSBs) are used. The 24 most significant bits (MSBs) are normally sign-extended. ARM-PA uses these 24 higher bits to hold information that will allow it to verify the pointer's integrity, called a PAC (Pointer Authentication Code). Note the actual size of the PAC depends on the system configuration and may be only 3 bits wide. The more bits available for the PAC, the more secure it will be. To do so, three new instructions are available.

'PAC* pointer-reg, context-reg' will compute the PAC of the given pointer and place it in its higher bits. The second operand is a context. It is used in the PAC computation, such that the pointer can not be decoded without the same context value. Without this context, an attacker could replace an authenticated pointer's value with any other authenticated pointer's. After this call, the pointer can not be used as a jump destination until one of instructions 'AUT*' or 'XPAC*' is called. If it is used in an irregular way, it is guaranteed that it will raise an exception.

'AUT* pointer-reg, context-reg' will check the validity of the PAC in the pointer. If the PAC is valid, then the register will contain the initial, usable pointer value. If the check fails, the register will be set with a value that is guaranteed to generate an exception if it is used as a jump destination.

'XPAC* pointer-reg' will just remove the PAC of the pointer without first checking it. It makes it usable as a jump destination.

The '*' in these instructions should be replaced with one of 'IA', 'IB', 'DA', 'DB' which determines which key is going to be used for the PAC computation. The processor is indeed capable of using up to four keys, that are stored in registers. Those registers are not available in user mode (EL0). It is expected that the key to secure a program running at a certain privilege level should be managed by the higher privilege level. The keys should also be short-lived and regenerated for each execution since otherwise a bruteforce attack would be possible.

A special use case, and a good way to demonstrate how ARM-PA works is protecting return adresses. When a function calls another one, it uses the instruction "jump and link" which saves the return adress in the link register. If multiple function calls are nested, this register should be temporarily saved in memory. ARM-PA can be used to protect this pointer which can be rewritten by exploiting e.g. buffer overflows. To achieve this protection

mechanism, one would put this at the beginning of the function, to encode the link register using the stack pointer as a context value.

At the end of the function, before returning, one would then verify the pointer's PAC. If the verification fails, the pointer becomes invalid and the 'RET' will raise an exception. Otherwise, the pointer will be valid and the 'ret' will work normally.

ARM has added special pseudo-operations which are just aliases for these two instructions: 'paciasp' and 'authiasp' As said in the requirements part, GCC 7 (or greater) supports automatically adding these instructions in each function's prologue and epilogue using the option '-msign-return-address'

ARM-PA is also capable of protecting data, altough it is not its goal. Using a fifth general purpose key called 'GA', the instruction 'PACGA rd, rs1, rs2' will produce a 32 bits wide PAC based on the content of the given two 64 bits registers. The PAC is placed in a destination register, and is be saved in memory. There is no 'AUTGA' instruction. To validate the PAC, one should regenerate it using 'PACGA' and check for equality with the previously computed PAC.

This security mechanism is not new nor better than other existing mechanisms, as it is just like computing a MAC to sign any data. Its main advantage is to be easy to use in a program targeting ARM-PA as it re-uses the same hardware and software.

The new instructions need to compute a MAC, which is rather slow. A SoC designer can choose to change the cryptographic primitive used, but ARM recommands its new QARMA as it is lightweight and made to support ARM-PA constraints. Typically, calls to 'PAC*' and 'AUT*' will each add an additional overhead of 6 to 8 cycles. Based on Liljestrand et al., the overhead is insignificant when signing return adresses and function pointers with roughly a 0.5% increase in execution time. This is due to the low amount of function pointers and return addresses (only one) in each function. Data pointers are much more frequent, and can thus induce a much higher overhead (19.5 % average in the benchmarks used).

Note that the only cost is that of the call to 'PAC*' and 'AUT*'. The overhead will thus vary greatly depending on the frequency and amount of authenticated pointers. One should make a compromise between the security level to achieve, and the acceptable performance overhead.

Trusted Execution Environment

A *Trusted Execution Environment* (TEE) [33] is a secure zone in the processor, where the code and data are protected in confidentiality and integrity. It is a way to isolate an execution thread. In a way, a TEE is more secure than a rich environment, while offering more functionalities than a *secure element* (SE). There are two industry standard organizations who work on the TEE:

- GlobalPlatform, which aims at standardizing the TEE specifications (a protection profile has been written) ;
- Trusted Computing Group, which tries to align the notions of TEE and of *Trusted Platform Module* (TPM).

A TEE can rely on a technology such as the ARM TrustZone, but this is not compulsory. An enclave in the SGX technology is a TEE. The main applications of the TEE are related to the protection of sensitive data, such as mobile payment or valuable multimedia content management in *set-top boxes*. The hardware implementation of this countermeasure is very much dependent on architectural choices and on requirements in terms of security.

The two next sections will be dedicated to the description of widely deployed TEE hardware implementations: TrustZone and *Software Guard Extensions*.

TrustZone

TrustZone [10] is a security extension from ARM. It allows to consider a unique processor as two virtual processors. The offered functionality is a switch between the two hardware *pipelines*. Dedicated hardware enables a complete isolation of the two virtual processors. In ARM terminology, these are called two 'worlds'. The notion of world is independent from all the other capabilities of the processor. For example, it is possible to run a rich OS in one world and security functions (of reduced size, and, thus, better checked) in another world, completely isolated from the rich one. For example, Samsung's Knox uses TrustZone to guarantee the kernel integrity.

Let's note that the ARM TrustZone technology was adapted by Trusted Foundations Software, which was acquired by Gemalto.

Giesecke & Devrient have implemented a comparable solution. The three technologies were grouped together in a *joint venture* called Trustonic (pun on: "Trust on IC").

However, implementation details of TrustZone being confidential, its robustness may not be evaluated, although it is well known that realizing attacks is possible, as shown for example by [46] and [65].

Please note that these two last references are about software bugs which allow to bypass the security of the trust zone.

Software Guard Extensions

Intel *Software Guard eXtension* (SGX) [37] has appeared in 2014 with the advent of Intel Skylake architectures, in parallel with the protection system *Memory Protection Extensions* MPX. This protection system is an extension of Intel architectures enabling to guarantee the security of applications integrity and confidentiality even when privileged software (kernel, hypervizor, drivers, etc...) are considered as potentially malicious.

This sytem allows to create applications which, when running, are isolated in a memory enclave, enabling the protection of their code and of sensitive data.

As shown in Figure 2.16 (page 79), the attack surface of the SGX protected application is in this way reduced and all attacks coming from the OS and the VMM (Virtual Memory Manager) are not possible anymore.

This protection is both at the hardware and the software level. At the hardware level, an instruction set is added to enable the switch between the standard software execution and the enclave execution. Activation generally takes place generally in the BIOS. At the software level, it is necessary to use a software development kit (SDK) when developing the application to be protected. The SDK allows the application to create an enclave in the memory, which enables its execution. This enclave is allocated and managed directly by the processor. It contains code and encrypted data, of which integrity is also checked. The processor contains a unique master key per

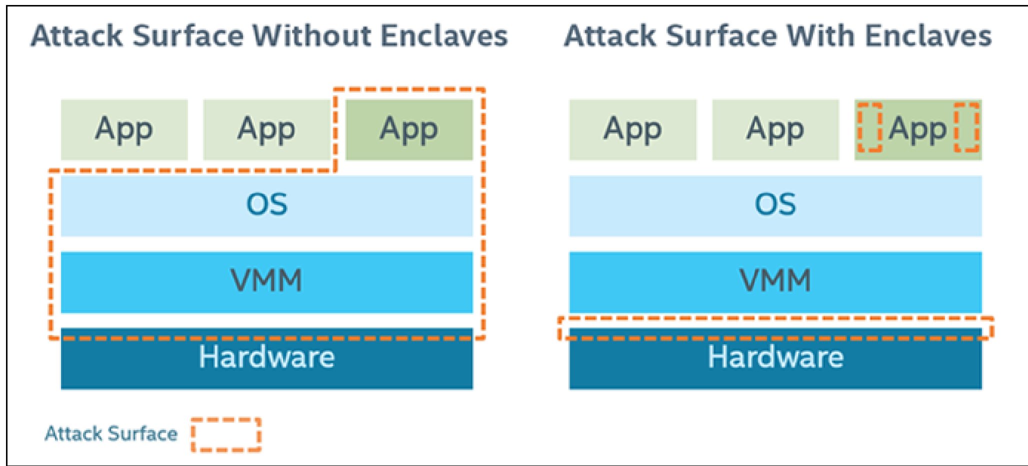


Figure 2.16: Attack surface with and without the SGX protection (diagram extracted from Intel website)

system, which is derived (instruction EGETKEY). There is no SGX *debug* mode at production stage. Practically, the application starts normally and then loads encrypted data in memory, after their validation by the processor. Then, the application starts in protected mode by running the code in memory and when it exits this mode, it may not access these protected data anymore. The Intel SGX system includes an attestation mechanism allowing to indicate that an enclave is set on a platform. The attestation may be:

- **local** – Two enclaves authenticate one another on the same platform, which is very useful when several enclaves must work together or when two applications must share sensitive data from the same enclave. Once the two enclaves have proved they could be trusted, they establish a protected session via an ECDH exchange to obtain a session key. This key then enables data encryption;

- **remote** (cf. Figure 2.17) – An enclave is recognized as trusted by a remote system. A third party equipment then enables trust establishment between two enclaves. For each platform, a hash of the software information is combined with a unique asymmetric key stored in the hardware, this message being sent to the third party equipment via an

authenticated channel. The third party equipment determines if the enclave is running on a Intel SGX processor and grants access to secrets it has selected via the authenticated channel.

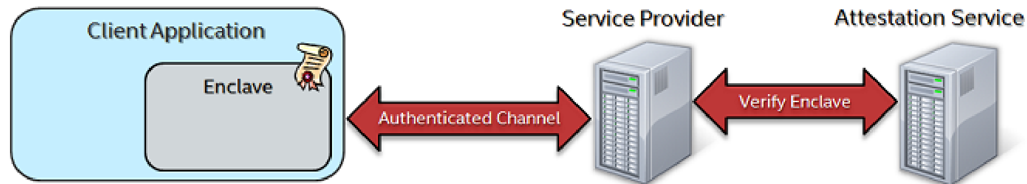


Figure 2.17: Remote attestation (diagram from the Intel website)

The SGX system also includes a *seal* function which allows to encrypt and save sensitive data that may be seen only when the trusted environment is restored. This function uses a hardware encryption function (integrated in the processor).

In theory this system seems to effectively protect sensitive data. An attacker might, nevertheless, use these enclaves precisely to store malicious code.

An open platform OpenSGX [66] was created in order to emulate the SGX system and perform research and tests. It allows to obtain a performance evaluation of such a system. Thanks to this platform, one can observe the number of cycles needed to access the enclave is exponentially related to its size. However, enclave initialization requires about 120M cycles, independently from its size.

Secure Memory Encryption

AMD's *Secure Memory Encryption* (SME) [8] dates back to 2016 with the advent of AMD Zen architectures. This protection system is a *Secure Processor* enabling encryption of data before it is copied to RAM. The impact of encryption on performance is low according to AMD, because it is possible to encrypt only parts of the data.

AMD associates this system to the *Secure Encrypted Virtualization* (SEV) which allows to control multiple virtual machines with the hypervisor. The

architecture of the SEV system is schematically represented in Figure 2.18 (page 81). Each virtual machine has its own encryption. The SEV system (memory encryption) is included in SVM (Security and Virtual Machine) systems which will be described in section 2.2.1.

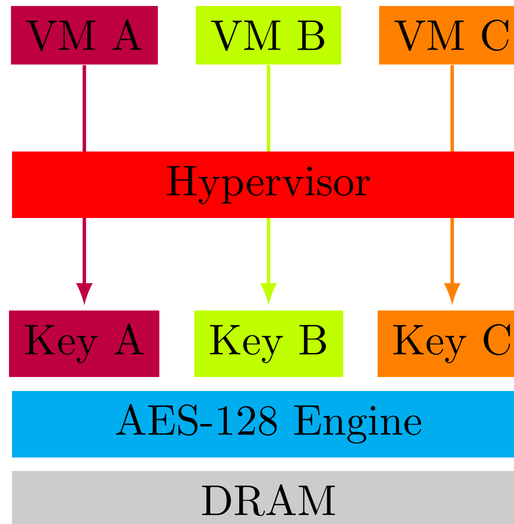


Figure 2.18: SEV Architecture (diagram inspired by the AMD website)

From the security point of view, this system can allow two users of the same station to have distinct encryption keys and, thus, the administrator does not necessarily have access to all the data of that station. The SME system uses the AES 128 bit encryption algorithm. The key is unique per system and is generated randomly at each start up. Moreover, this encryption key may not be read or modified from the software applications. Each individual page of the memory pages' table is marked as encrypted or not. As shown in Figure 2.19 (page 82), each data read from a page marked as encrypted must automatically be decrypted and each data written in a DRAM page marked as encrypted must be automatically encrypted/decrypted. This system, thus, enables encryption from startup, allowing, for example, to have an encrypted kernel.

The CPUID opcode (instruction for x86 architectures) is used to check if this protection system is available or not. At start up, the system is activated via the SYSCFG MSR (syscfg : system configuration tool of the *firmware* and the BIOS). Then, the C-bit is used to check if the memory page is

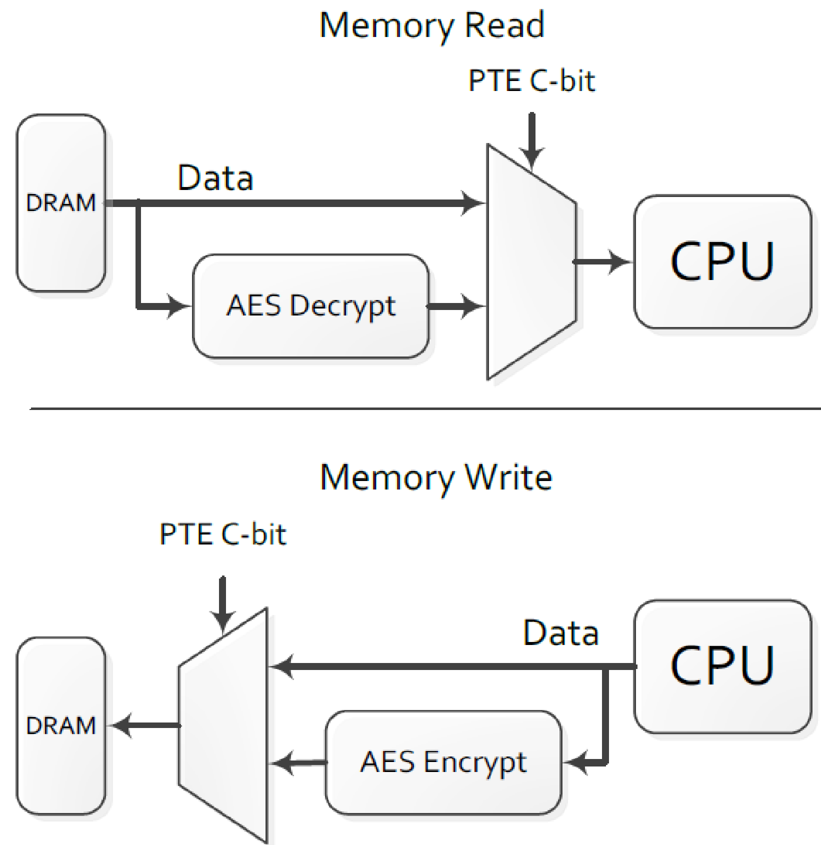


Figure 2.19: Read and DWrite in SME mode (diagram from the AMD website)

encrypted or not. The C-bit location (by default bit 47) is also determined by the CPUID.

The impact on performance is minimal, only a small additional latency appears for encrypted pages. Encryption depends upon the address, which allows to prevent attacks based on data /code displacement (so called *splicing* attacks). The algorithm used is AES-128 and the key is managed by a microcontroller. The operating mode of the AES is not indicated. One may think it is ECB, but OFB is also possible (which would allow to absorb the encryption time in the AES calculation latency).

Contrary to Intel SGX system, the application does not require any

change, so this system is easier to implement. The encryption key management is completely performed by the hardware system (AMD Secure processor). This system protects against physical attacks on memory. Unlike the Intel SGX system, it does not protect, for example, against an attacker who compromised the kernel.

It protects mainly against:

- *Cold-boot attacks* (Attacks by cold start allowing to recover encryption keys from a hard disk via an auxiliary channel, requiring a physical access to the system);
- *snooping* on the memory bus ("sniffing" sensitive data on the bus);
- uncovering of temporary data saved in the persistent memory.

Virtual-Machine Extensions

Virtual-Machine Extensions (VMX) from Intel is an addition of instructions to the processor in order to enable hardware assisted virtualization. A schematic representation of this method is shown Figure 2.20 (page 84). As was explained in the section about software countermeasures in the previous report, virtualization is used to isolate services. As this technology is used very often in the industrial market, it was very quickly implemented in hardware. The addition of virtualization capabilities to the processor enables to simplify the *Virtual Machine Monitor* (VMM) and to dramatically increase performances compared to pure software solutions. To this end, new instructions are added to the processor. Each process, launched under VMM control (thanks to the command `VMLAUNCH`), considers it is the only process running on the processor.

Recently, the extension *Extended Page Tables* (EPT) has been added to new generation processors which allows each virtualized software to manage their own table page, has been added to new generation processors. Before the implementation of this extension, the VMM was in charge of addresses translations.

Regarding performances, complete OS virtualization is almost as efficient as having a real machine, and eases the sharing of resources.

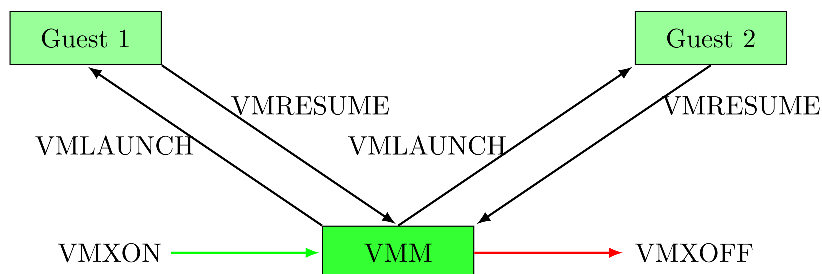


Figure 2.20: Schematic representation of VMX

Secure Virtual Machine

Secure Virtual Machine (SVM) from AMD is a hardware system improving the performances of virtualization systems on x86 architectures. This system, which is similar to the Intel VMX solution, was introduced in 2004 under the name "Pacifica" [5] and is now known as AMD-V. The first processors implementing this system were introduced in 2006.

Virtualization uses a *Virtual Machine Manager* (VMM) which is between the OS and the hardware, and which allows to run several systems on a unique physical machine. The AMD-V is a hardware extension which accelerates the management between a virtual machine and the hardware.

In order to run, it must be activated in the BIOS. Like the Intel solution, it is made of a new instruction set (`VMRUN`, `VMLOAD`, `VMSAVE`, `VMMCALL`, `STGI`, `CLGI`, `SKINIT`, `INVLPGA`). These commands allow the VMM (hypervisor) to enter/exit the guest mode of the processor. The AMD technology for the management of the *Second Level Address Translation* (SLAT) mechanism, which is equivalent to the *Memory Management Unit* (MMU) on virtualized systems, is *Rapid Virtualisation Indexing* (RVI) (see Figure 2.21).

This mechanism enables the protection of the memory pages of the hypervisor and the memory accesses between VMs. There are three different types of addresses:

- virtual addresses related to the guest process (converted to physical addresses for the VM by the MMU);
- physical adresses related to the VM (converted to physical addresses

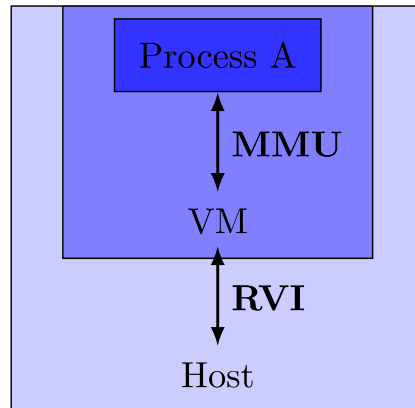


Figure 2.21: SVM protection representation

for the host by the RVI);

- physical addresses related to the host.

Silicon Secure Memory

Oracle, in its SPARC M7 processor, proposes the *Silicon Secure Memory* (SSM) [55] protection, which performs a hardware check of pointer bounds during execution.

After the memory zone is allocated, a code is attributed to that zone and stored in the pointer, and when the pointer is used to access the memory, the hardware checks if the pointer code is identical to the memory zone code. This code is also called *'colour'* because this technique has a lot in common with *tainting*.

In order to use this countermeasure, the application must respect some conditions, which are:

- it should be be compiled in 64-bits;
- SSM for the targeted memory zone should be enabled;
- the allocated memory zone must be aligned on 64 bytes;

- the size of the allocated memory zone must be a multiple of 64;
- the memory zone must have a colour;
- the pointer must have the same colour as the zone it points to.

A synthetic representation of this countermeasure is shown Figure 2.22.

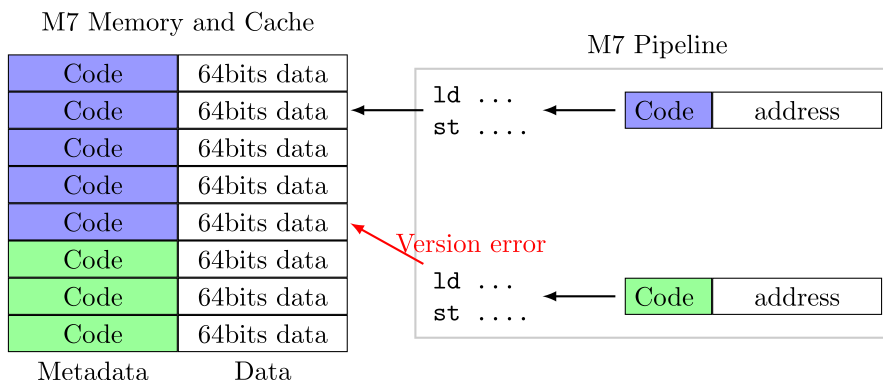


Figure 2.22: SSM representation

There is not much information available on this technology. According to the Oracle blog [60], a subset of the pointer bits is used to store the code, which explains why the memory zone must be aligned within 64 bytes. But there is no information on the way the allocated zone colour is stored and where, or how two colours are compared by the processor.

The probability to detect a non-aligned memory access is proportional to the number of bits used to store the code.

The table below shows the detection probabilities as a function of the number of bits used to code the colour, based on the hypothesis that the colour is chosen randomly.

Number of used bits	Detection probability
1	50%
2	75%
3	87.5%

If the colour is not chosen randomly but selected by the memory allocator in order to systematically have a colour different from that of adjacent regions, then the detection rate is close to 100%. This countermeasure can also prevent temporal errors by ensuring that a zone previously allocated does not reuse the same colour.

Regarding performances, Oracle declares that the *overhead* caused by the countermeasure is less than 1%.

2.2.2 Academic studies

In this section, we will present countermeasures proposed in the academic litterature. We will begin with the protections designed by Secure-IC and Télécom-ParisTech in the framework of the project RAPID CyberCPU. These countermeasures focus mainly on the protection of the CFG of applications. Other protections proposed in the literature, based on different methods, such as *tagging*, will then be presented.

Secure-Call

Secure-Call (SCALL) is a countermeasure developed by Secure-IC in cooperation with Télécom-ParisTech in the framework of the project RAPID CyberCPU. This countermeasure focuses on the protection of the *backward edge* of the CFG, that is function returns. To this end, the *pipeline* of the SPARC/LEON3 processor has been modified so that the instructions `call` and `jmpl`, which normally store the return addresses in the register `%o7`, also store them in a hardware stack called SCALL. This stack can only be accessed by the processor. The running software has no access to the memory of this *shadow stack*, thus, achieving a protection against the bypass of the countermeasure by an attacker. For the software, the management of this *shadow stack* is totally transparent.

When there is a function return, the address contained in register `%o7` is used. The comparator module of SCALL then checks if the two addresses, the one in the software data stack and the one in the *shadow stack* SCALL,

are identical. If these two addresses are equal, the module then sends back the return address. If they are different, the comparison module of SCALL raises an exception in order to warn the operating system that an anomaly was detected. It is then up to the OS to decide which security policy to follow. The Figure 2.23 (page 88) schematically shows the operation of SCALL.

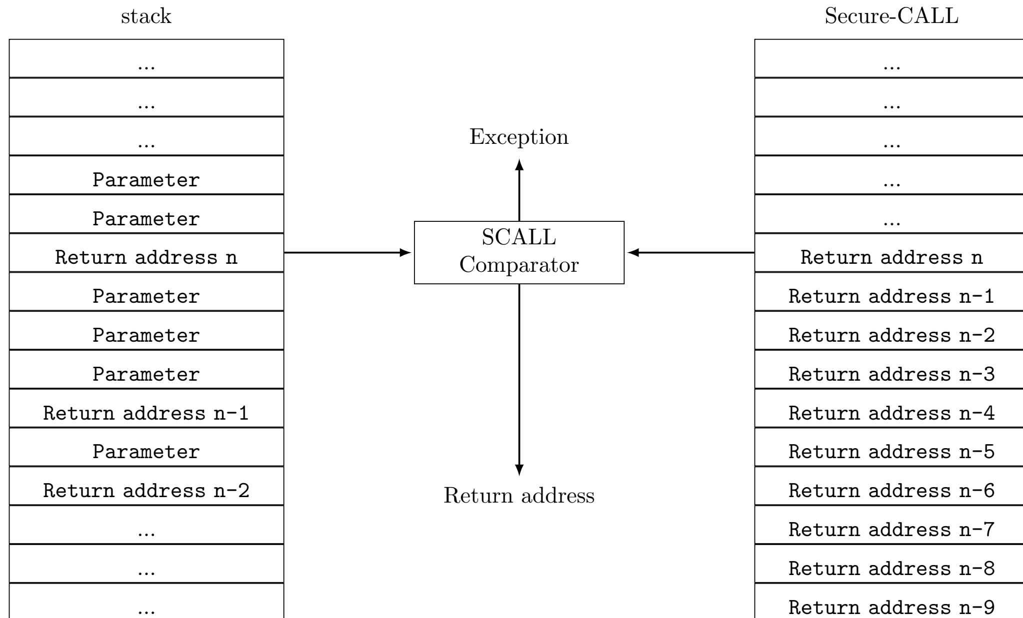


Figure 2.23: Schematic representation of the countermeasure SCALL

Three different security policies can be distinguished as:

- HARD policy: the process is simply aborted ;
- SOFT policy: the process continues to run – this may be useful in order to analyze an attack;
- SAFE policy: the process continues to run but uses the return address given by the *shadow stack* SCALL.

If there is an exception, it is up to the OS to correctly update the *Shadow Stack* so that no alarm is triggered at the return from the exception. Similarly, in case of *fork* or a programming *multithread* occurs, the OS copies the

contents of the *Shadow Stack* in the data of the new processes. If an `exec*` is executed, the *Shadow Stack* is then reset.

The *shadow stack* SCALL being sized to contain 64 return addresses, it is sufficient for the majority of software which were tested, knowing that they use, on average, only 14 entries in this stack. No noticeable decrease in performance was seen on the test *benchmark*.

HCODE

Like SCALL, HCODE has been developed in the framework of the CyberCPU project. This protection focuses on the check of the integrity of the *Basic Block* (BB) executed in the processor. First, one needs to identify the *Basic Block* in the software to be protected. To that end, the compiler is modified to insert labels at the beginning of each BB. To correctly run the protection, HCODE must be able to detect the ends of BB at runtime.

This is generally easy because a *Basic Block* usually ends with a jump instruction.

However in some cases such as the example shown in Figure 2.24 (page 89), the end of a BB can be associated with the fact that the next instruction is the target of a jump. In that case, the detection of the end of BB end is not straightforward at runtime. For this reason the compiler is modified so that all the *Basic Blocks* end with a jump instruction, even if it is a jump to the next address.

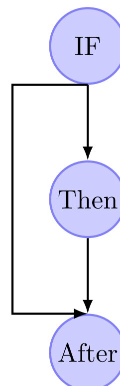


Figure 2.24: Example of *Basic Block (BB)* without jump in the end

The second step in the compilation is to calculate a HASH of the instructions for each BB. A list of triplets (BB start address, BB end address, BB HASH) is then generated for each *Basic Block* in the code and stored in a special section of the executable file.

At the time of loading the software in memory, the *loader* copies the section of the binary, containing the triplets, in a memory zone dedicated to the HCODE moduleso that the hardware module may access it. At runtime, the HCODE module calculates the HASH of instructions executed by the processor on the fly. When a jump instruction is executed by the processor, the countermeasure HCODE deduces that it is a *Basic Block* end. It then fetches the reference HASH from the memory, calculated during the compilation, and compares it with the one calculated during the execution. While the reference HASH is researched, the processor is paused, so that the HCODE module may fetch and compare the HASHs. If the two HASHs are the same, the processor is restarted. If the two HASHs are different, an exception is raised to inform the OS that an anomaly has been detected. Figure 2.25 shows the HCODE hardware module.

HCODE does entail a noticeable performance loss on software execution. This is mainly caused by the time necessary for the module to find the reference HASH. This research is, for the time being, performed in a linear and a better way search algorithm would improve the performance. Future HCODE developments will implement a complete software check of the CFG. To that end, the OS will be in charge of checking that the history of the executed BB is conforming to the CFG.

Project CHERI

Project CHERI (*Capability Hardware Enhanced RISC Instructions*), presented in march 2012, proposes the implementation of a countermeasure based on the use of *fat pointers*, which enables the detection of the *overflows* of a table. To that end, rather than representing a pointer with its address, a solution is to add the address of the first word in the table (*base*) and the end address of the table (*bound*) as shown schematically in Figure 2.26. This representation is exactly what is called a *fat pointer*.

During use, the *fat pointer* (*i.e.* the bits containing the addresses) may

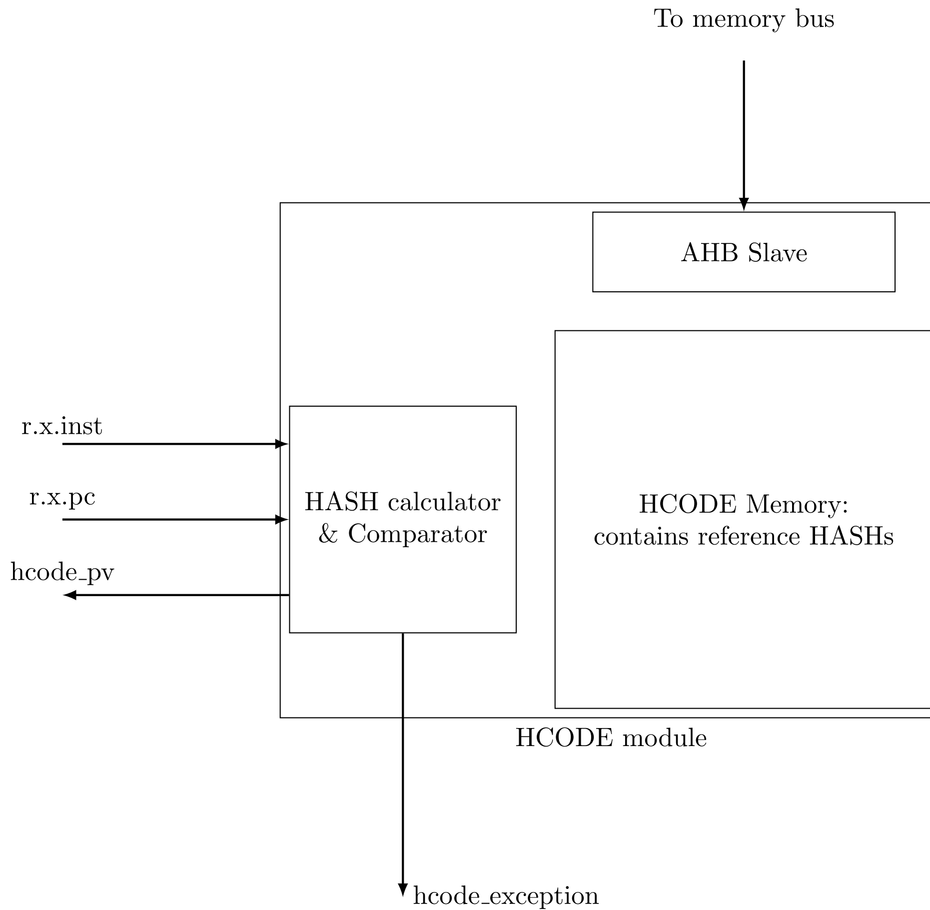


Figure 2.25: HCODE hardware module

be incremented or decremented. If the resulting pointer is still within the bounds, the software continues running. Otherwise an alarm is raised.

The main issue with this method is the huge cost in memory linked to the use of *fat pointers*. In the CHERI project, the *fat pointers* are mapped on 256 bits including 64 address bits (64 bits for the *base* and 64 bits of *bound* [72] [75]). Among the remaining 128 bits, 31 are used to manage permissions (write, read, run) and the last remaining ones are reserved only for research and tests. For industrial applications, it will be possible to eliminate some of these bits. To implement this protection a recompilation (done via LLVM) is necessary. Then, the management of these *fat pointers* is performed by a

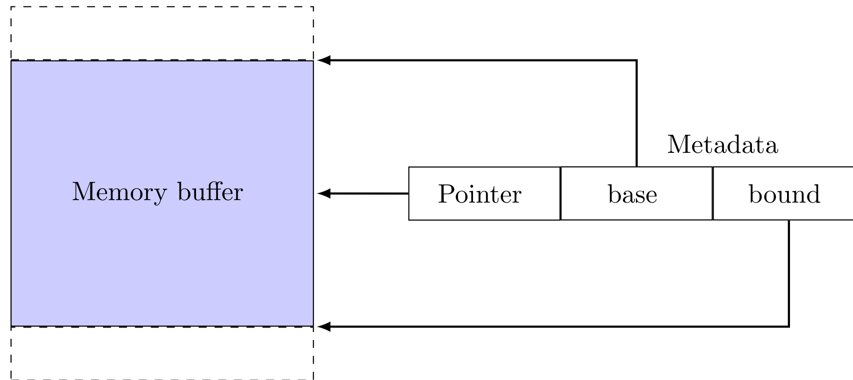


Figure 2.26: *fat pointer* representation

dedicated coprocessor.

Processor SAFELite: Low fat pointer

In [45], A. Kwon et al. propose an adaptation of *fat pointers* called SAFELite processor, which introduces a new representation of pointers, the *low fat pointers*. This solution is interesting because of its low cost in terms of gates count which allows a short calculation time. The encoding of *fat pointers* called "BIMA" is based on a floating representation of the segments size. For example, for this processor, the pointer and its associated metadata (*base* and *bound*)can be contained in a unique 64-bit word for a 46-bit address range as shown in Figure 2.27.

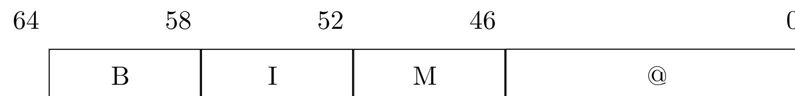


Figure 2.27: Diagram of a BIMA type counter

In this Figure, the various fields on the BIMA pointers encoding represent:

- B: the size of each block is 2^B words;
- I: the base is given by $2^B \times I$ words;

- M: the bound is given by $2^B \times M$ words.

In a case where :

- B = 2
- I = 1
- M = 4
- @ = 5

For such values, the various words in memory are shown Figure 2.28.

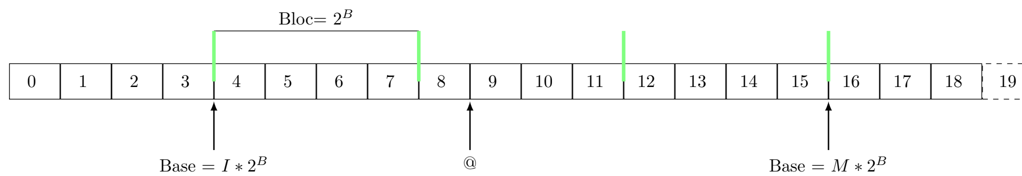


Figure 2.28: Example of *low fat pointer*

In the SAFElite processor, creation and use of *fat pointers* are managed by a special unit called *memory manager*.

Moreover, to ensure an enhanced security of data management, 8 additional bits of information about the type are added to the data.

HardBound

Devietti et al. propose in [29] a hardware solution in order to manage the spatial security of the memory for pointers in the heap and the stack, as well as global pointers. The target of this method is to optimize the cost in memory and in calculation caused by the *fat pointers*.

The principle of the HardBound architecture is to add 4 tag bits to data in order to identify their type:

- not a pointer;

- one among 14 types of compressed fat pointers: "tag";
- an uncompressed *fat pointer*.

In the case of compressed *fat pointer* with type "tag" and tag $= 4, \ldots, 14$, the *base* is equal to the pointer and the *bound* is equal to $4 \times$ tag. A *fat pointer* must be uncompressed:

- if its size is not a multiple of 4;
- if its size is over 56 bytes;
- if the pointer does not point to the start of the object.

Compilation-Enforced Temporal Safety

Many tools allow to detect and manage, more or less effectively, the *overflows*. Nagarkatte et al. propose a solution called CETS (Compiler-Enforced Temporal Safety) [50].

The principle of CETS is based on a "lock-and-key" identifier and on the use of a *shadow memory*. In fact, CETS gives each pointer two additional fields: a unique key and a lock address (`lock address`).

Figure 2.29 schematically represents the CETS operation. When a table

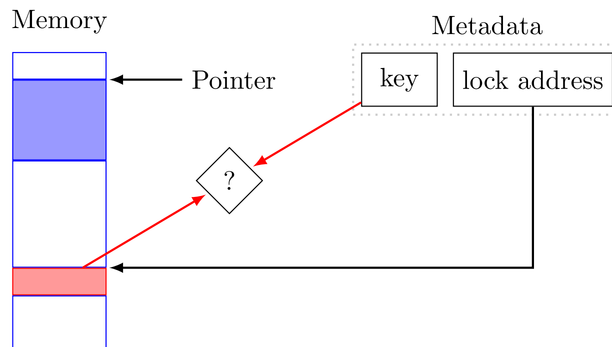


Figure 2.29: CETS schematic operation

(or another object type) is allocated, the pointer key (`key`) is set with a unique value. That key is saved in a memory area called `lock`. The address of this memory area is saved in the second additional field of the pointer `lock address`. When the pointer is deallocated the key (`key`) is reset.

For each use of the pointer, the key (`key`) and the lock value (`lock`) pointed by `lock address` are compared. If these two values differ, an alarm is raised because a temporal corruption of the pointer occurs.

Moreover, in order not to alter compatibility and to improve metadata security, the values (`lock` and `key`) are not placed after the data ("in-line metadata" like in *fat pointers*). They are stored in a *shadow memory* using a two-level sorting tree.

A sorting tree is a multi-level data structure in which each level can be accessed, thanks to different sets of input bits.

The impact of this countermeasure is an increase of the runtime by 39% when it is supported by hardware, compared to a 108% increase in case it is fully implemented in software.

Architectural Support for Instruction Set Randomization

Architectural Support for Instruction Set Randomization ASIST [56] is a modification of the LEON3 CPU which enables the dynamic encryption of executable code in RAM at loading as well as its on-the-fly decryption at execution. Only the code zone (`.text`) is encrypted, while data (`.data`, `.rodata`, `.bss`, `.stack`, `.heap`) are stored unencrypted. The encryption key is dedicated to a given executable code: it is loaded in the processor registers when there is a context switch, that is, when the multitask operating system scheduling indicates it is time to run another process. Related libraries are also encrypted with the key of the executable, as well as the return addresses (in the stack zone), because they are considered as code chunks which the user should not be able to access. The `.text` zone encryption prohibits all injection code attacks: the injected code would have to be encrypted by the attacker before injection, since the CPU would decrypt it automatically in order to execute it. But the encryption key is randomly generated at each software execution, and the encryption algorithm is considered secure against the attacker's limited capabilities. When considering such hypothesis, a XOR

with the key, a XOR with several keys derived from the master key, or a permutation controlled by a key are considered as secure. Such encryption does not protect against code reuse, so encrypting also the return addresses can be used to prevent an execution flow modification by controlling the return address. The attacks which are still possible are attacks based on code reuse which do not rely on suppressing the return address of a function, such as a JOP. The runcode encryption mechanism is dynamic and is performed when loading the code (when it is copied from the HDD to the RAM), so that ASIST may function without code recompilation.

In terms of silicon area, this countermeasure increases the used area by 7% maximum. The impact of the countermeasure on applications' runtime is negligible.

In [25], it is proposed to add, to the processor, a mode in which the instructions fetched in memory are encrypted. Thus, the processor needs to decrypt before executing them. This countermeasure is intended for software protection against *reverse-engineering*, but an interesting aspect of this kind of countermeasure is that it also limits the number of gadgets an attacker may use to perform a *code reuse* attack. Indeed, contrary to the ASIST countermeasure, REV [25] allows, thanks to specific instructions, to switch between encrypted and clear modes. The attacker, thus, cannot use encrypted code when in clear mode, and is obliged to use encrypted gadgets only in encrypted mode, and unencrypted gadgets only in clear mode.

Software and Control Flow Integrity Architecture

Software and Control Flow Integrity Architecture (SOFIA) [27] proposes to couple the control flow integrity (CFI) and software integrity (*Software Integrity* SI) protections based on ISR.

To that end, the software is divided in a set of encrypted instructions blocks containing a checksum for these instructions. The flow integrity is checked with the decryption part, the key used to decrypt the current instruction being a combination of its address and the address of the previous instruction. So, if the CFG is followed, the instructions are decrypted correctly.

On top of the encryption, each instruction block includes one (or several)

checksums corresponding to the instructions in the block. A check is performed at execution: if the two checksums are different then the controller is re-started. Figure 2.30 (page 97) schematically shows the operation of this countermeasure. In order to execute a non-linear flow, the concept of *Basic Block* with several predecessors is introduced (see Figure 2.31, page 98).

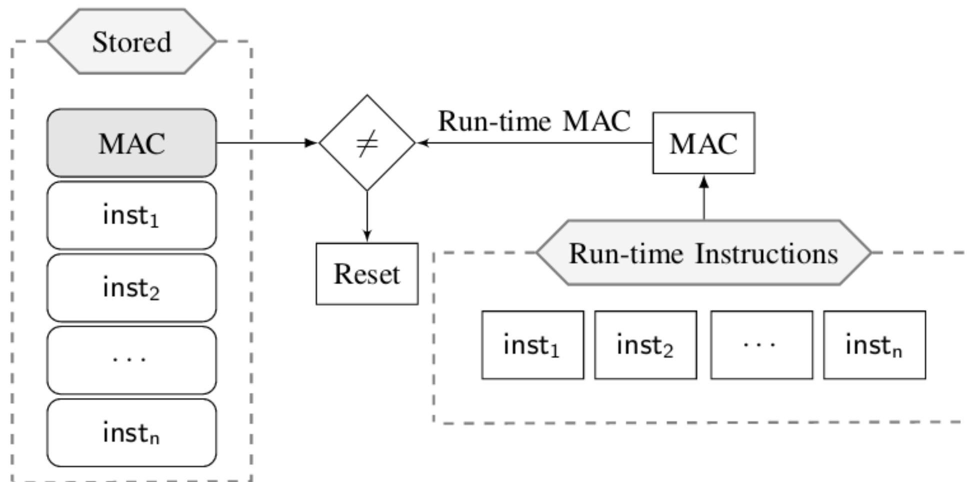


Figure 2.30: SOFIA operation diagram (extracted from [27])

This countermeasure was implemented on a FPGA with a LEON3 processor. An increase in 28.2% of the used area has been measured. The decrease in performance is mainly due to a decrease of 84.6% in the clock frequency and to an increase in the execution cycles by 13.7%. Overall, the total execution time increases by 210% mainly due to changes made inside the pipeline of the processor.

On top of this significant performance loss, this protection is not designed to work on complex systems with an OS. Moreover it is not able to protect complex or dynamic control flows. So it is not possible to protect code using polymorphism, because it is mandatory to know all the branches of the CFG at compilation time, which is not possible with the polymorphism principle.

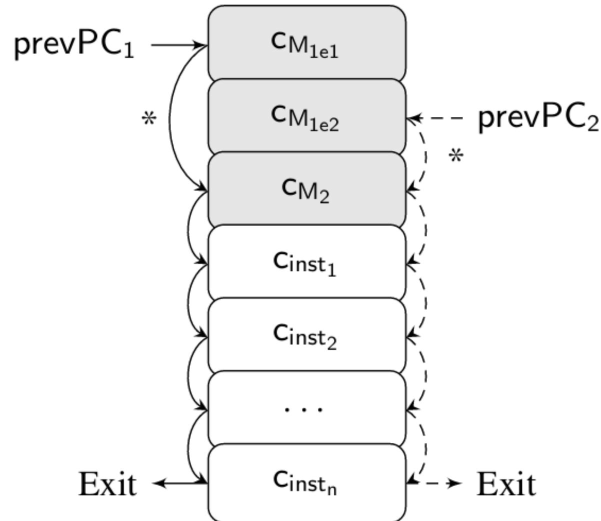


Figure 2.31: *Basic Block* allowing two predecessors (diagram extracted from [27])

Hardware-Assissted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation

[26] propose an hardware implementation of Abadi's paper [3] [4]. For this purpose, two new instructions are added: `CFIBR`,`CFIRET`. And the `RET` instruction is modified, or the new `DEACT` instruction is implemented.

Because this implementation can only raise an alarm on wrong return label, it is not capable of detecting JOP attack, which never use return instruction. To address this problem they use counters to monitor executed instruction in the windows of five indirect jump. This give indicator that a JOP or a ROP attack is undergoing.

2.2.3 CFI

Previous works on control-flow integrity (CFI) are generally based on software instrumentation implemented by the insertion of additional instructions on each block [34, 4, 47].

Let us describe more specifically three existing representative solutions which implement CFI:

- Enhanced Control Flow Checking using Assertions (ECCA [7]);
- Control Flow Checking by Software Signatures (CFCSS [53]);
- Hardware-Assisted Fine-Grained Control-Flow Integrity (HAFGCFI [26]);
- Hardware-enforced Control Flow Integrity (HCFI [18]).

ECCA

ECCA is able to check the correct execution of the control flow by assigning an unique prime number for each basic block. Each basic block is decorated with two new code-checkers:

- At the beginning: with a *test* assertion which checks whether the previous basic block is permissible.
- At the end: with a *set* assignment which updates an identifier taking into account the whole set of possible next basic block.

This technology is subsequently tested using automated fault injection benches, and most of the contribution is geared towards the simulation-based fault coverage methodology.

CFCSS

CFCSS assigns a unique signature to each basic block and, for the purpose of checking, a global variable contains the run-time signature. But CFCSS cannot cover control flow errors if multiple nodes share multiple nodes as their destination nodes [53].

These two first previous works have some issues: they suffer from memory overhead and low performance due to the large size of the protection code

added for checking flow execution. We will show that hardware / software collaboration allows for better performance.

Some other CFI protections use hardware for improved performance [26], as discussed here-after.

HAFGCFI

HAFGCFI is based on checking every function call. Each function is associated to a unique label and each call to the function is replaced with a new CFI instruction called `CFIBR`. A shadow stack is used to save the call-tree containing the function label called by `CFIBR`. Also each return instruction is replaced by a `CFIRET` which checks the consistency of the return (the label of the `CFIRET` is in the call-tree), thereby preventing ROP and JOP. But for indirect call and jump, an heuristic behavioral-based approach is used by analyzing number of `POP`, `PUSH`, and indirect jumps.

Like other software implementations it checks transition upon "jump" instructions but does not verify the integrity of the executed code.

CFI can be also used for enforcement of more elaborate security policies such as Inlined Reference Monitor (IRMs) and Software Fault Isolation (SFI) [31, 32]. This use of CFI in these securities can help to prevent an attacker from bypassing the control sequence.

HCFI

In paper [18], the authors present HCFI (*Hardware-enforced CFI*), which is a modified SPARC architecture. It combines a shadow stack with a CFI-dedicated extension of the SPARC instruction set. While the solution concentrates only on call/return instructions, it achieves an impressively low run-time overhead of only 1%.

The sequel of this manuscript presents a solution which is further optimized (based on caches), processor-agnostic, and covers a larger number of faults (including physical disruptions, as discussed in the next chapter).

Chapter 3

Threat Model

3.1 Statement of threat model

We consider a powerful attacker able to perform both software and hardware attacks. In other words, the attack vector can either the exploitation of an existing bug (which allows to corrupt the state of the program) or an alteration caused by external stress which modifies the internal state. Figure 3.1 illustrates some emblematic attacks which are detected by our protection called CCFI. The hardware attacks are detected if the perturbation targets either the code or pointers to the code. Notice that some recent attacks (such as RowHammer and PlunderVolt) are software-triggered hardware modification attacks, hence are depicted inbetween software and hardware attacks.

As physical attack he is able to modify one word in memory, bypassing hardware memory protection like `W^X`. With this he is able to modify instruction in the code in order to skip an instruction or to change an instruction to another one.

As software attack he is able to use any vulnerability present in the software. This includes:

- Using buffer overflow on the stack to change return address
- change function pointer

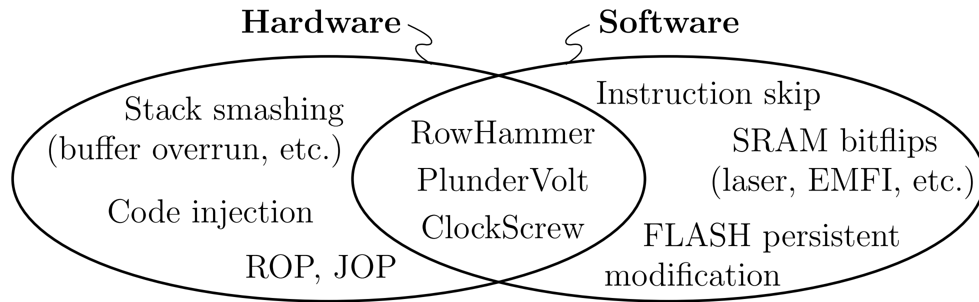


Figure 3.1: Attack vectors addressed by our CCFI technology

3.2 Mitigation of the threat

To protect against the aforementioned threats we implement CCFI (see chapter 5). In this respect, we must introduce several technologies:

- Basic Block (BB) verification (HCODE), as described in chapter 4;
- CFG verification, including forward and backward BB transition, as described in chapter 5;
- Support of interruptions and OOO execution, as described in chapter 6.

3.3 Offered security

3.3.1 Fine-grain security

CCFI offer the most fine-grained CFI security possible at instruction granularity. Its construction of check at the end of BB before jumping anywhere make it impossible for an attacker to modify pointer (function pointer or return address) by any mean to jump on his code or on gadgets.

3.3.2 Reactive vs infective strategies

Other CFI protections or protections like ASIST [57] offer resistance of the code against data injection by obfuscating those data in memory. Though such strategy is certainly effective against a vast majority of assaults, it suffers from a lack of reporting capabilities. Indeed, upon attacking ASIST, the process or the OS will crash, but no further information about the perpetrated penetration is available.

Still worse, before crashing, the program will execute inconsistent code, which can result in random corruption(s) of the memory. Now, it is known since 1997 that errors in cryptographic implementations can lead to full key extractions. For an overview of fault-based cryptanalysis, we redirect the reader to the reference book [40]. Of special interest are fault-based attacks which work even if the fault falls at a random location. We detail hereafter two examples: one in asymmetric and another in symmetric cryptography.

1. *In asymmetric cryptography*, the BellCoRe attack [15] shows that if whatever is broken in a computation of CRT-RSA, then the modulus N can be factored in its two prime factors p and q (provided the input of RSA is known). Initially, the authors of the BellCoRe attack imagined hardware faults, such as “hardware glitches that cause the processor to miscalculate”, or “induced faults”. The case of latent faults, i.e., when the hardware itself is computing wrong, has been developed by Biham, Carmeli and Shamir [12]; it is known as the bug attack.

2. *In symmetric cryptography*, the “differential fault analysis” paper by Biham and Shivam [13] shows similar results. In the section 5.1. of this article, it is explained how some bits of the secret key can be guessed provided one bit of a register is permanently set to zero. Now, the same effect can be obtained if a some entries of the substitution boxes of the block cipher are erased.

So, in both asymmetric and symmetric cryptography, alterations of the code or the data can lead to efficient cryptanalyses. Now, it can well be that the fault results from a memory corruption caused by the crash of ASIST. We are not aware of any proof-of-concept of such attack, but it is plausible. Thus, *cyber-induced fault attacks* must be prohibited, for instance by a suitable detection method.

This is where CFI (and in particular our hardware-assisted approach) turns out to be useful. Indeed, in case of a security violation, the system can log the event, which enables for statistics. A kernel module can be easily designed to handle such reporting. Such report can be made very accurate, including for instance the address at which the program started to go astray, and a memory dump (core file). So, even forensic replay of the attack becomes possible. But more precious information can be gathered from the attack event:

- the system can proactively stop and open a debugger console, allowing the user to get direct access to the attack being executed (useful in honeypots);
- the system can take timely actions, such as killing the attacked process.

3.4 Comparison with SOA

As we have seen in chapter 2, lots of existing counter measure implement fat pointer technique to ensure memory safety (MPX 2.2.1, SSM 2.2.1, CHERI 2.2.2, SAFELite 2.2.2, HardBound 2.2.2 and CETS 2.2.2). On the figure 2.14 we can place these protections on the first line, they prevent to `"Make a pointer go out of bound"`.

While other protections focus on protecting pointer with Code Pointer Integrity protection (CET 2.2.1, Shadow Stack 2.2.1, Pointer Authentication 2.2.1 and Secure-CALL 2.2.2).

Our solution protect any code modification by verifying an hash for each executed basic bloc. This protect against code corruption attack witch can be made by software or by fault injection. This Code Integrity protection is on the third line on figure 2.14 as `"Modify code ..."`. The second part of the CCFI protection implement Control Flow Integrity preventing to `"Use pointer by indirect call/jump"` and `"Use pointer by return instruction"`, fifth line of the figure 2.14.

In summary our protection able to prevent `Code corruption attack` and `Control-flow hijack attack` of the figure 2.14.

Chapter 4

Functional Mechanism of Control Flow Integrity

4.1 Formalism

4.1.1 Definitions and security property

In this section, we define the protection of the program flow and its associated instructions.

Definition 1 (Program)**.** A program is an oriented graph, where vertices are addresses (values of the program pointer, in the sequel referred to as "PC"), and where the edges represent the allowed transitions.

Dynamically, the execution of a program is denoted as a *program flow*. It consists in a walk on the oriented graph.

We notice that it could have been possible to define a program by labelling vertices with instructions instead of addresses. But actually, it amounts the same by the introduction of a program binary code:

Definition 2 (Binary code)**.** The binary code of a program is a table in which every licit address of the PC is mapped to an instruction.

This notion corresponds to the .TEXT segment of a program object, where the possible addresses translations are ignored. Therefore, the notion

of *running code* is simply the mapping of the sequence of addresses from the *program flow* through the binary code. In the sequel, we denote by T this translation table, between addresses and instructions.

Notation 1. *We denote by* $T[@]$ *the instruction in the binary code at address* @.

We attract the reader's attention on two limitations of our modelization:

1. The data are not modeled, hence errors on data are not captured. Other protections for the data will be needed on top of HCODE, such as *dynamic information flow tracking* [67]. In particular, if a data, whose purpose is to select amongst various licit transitions, is corrupted, then the program flow will be considered correct (despite it is not!), in the sense than no new edge is added to the graph.

2. The program graph is known statically, in the sense that no transition can be created dynamically. This is a limitation of our approach on the input language, compilation and link tools. One idea to get over this limitation is to make our CFG early in the compilation chain. If the CFG is deduced from a high level language, it is possible to catch the construction of dynamic calls, like switch tables and `vtables`.

A convenient representation of a graph is a list of vertices, each of which being the address stored in the vertex and the list of next (downstream) vertices.

Definition 3. For every address @, let Succ[@] denote the list of successors of @ in the program graph.

We can also define the predecessors of an address in the program graph:

Definition 4. The predecessors $Pred[V_1]$ of vertex V_1 is the set defined by $\{V_2 \in \text{program flow graph}; Succ[V_2] = V_1\}$.

Under nominal execution, the dynamic program flow remains strictly within the program graph. Similarly, under nominal execution, the running code corresponds to the image through the binary program of a licit walk in the program graph. This is formally expressed by this property:

Property 1 (P: unaltered execution)**.** *An unaltered execution (property named P) consists in checking both that:*

1. *the dynamic sequence of PC addresses belongs to the static program graph, and*
2. *the dynamic sequence of instructions belongs to the image of the set of dynamic sequences on the static program graph through the binary code T. Equivalently, there exists a dynamic sequence of PC addresses belonging to the static program graph such that its image through T yields the dynamic sequence of instructions.*

Here are some examples of attacks that would violate the property P:

- Jump at an address which is not a successor of this vertex. This can be due to a classic program flow hijacking, for instance by stack smashing: the return address of functions are overwritten. Another example is return- or jump-oriented programming (ROP & JOP).
- Replace code, which is possible, for example if the hardware does not support RO (read-only) restriction on the executable portions of the memory (.TEXT section for instance), or under debug. Another possibility is when laser, electromagnetic fault injections or SEUs occur.

In such cases, an attacker would violate the property P.

4.1.2 Concept of seamless verification of security property P

The figure 4.1 represents the different finite state machines (FSM) involved in the verification of property P. The left-hand side represents the functional FSM of the execution platforms, namely the update of the program counter (PC) to the next instruction address (nPC), and the update of the opcode register (denoted by “instruction”). The right-hand side are the FSMs which are added for the purpose of verifying the security property P. A copy of the program graph is stored securely, so that for every value of the PC, the list of licit successors is known. At every execution step, the hardware verifies

that the sequence of values of the program counter remains on the program graph. This is the implementation of the runtime verification of the first item in property P. A sibling FSM checks the sequence of instructions. For this purpose, a copy (T_copy) of the program binary (table T) is used for sanity check. This implements the runtime verification of the second item in property P.

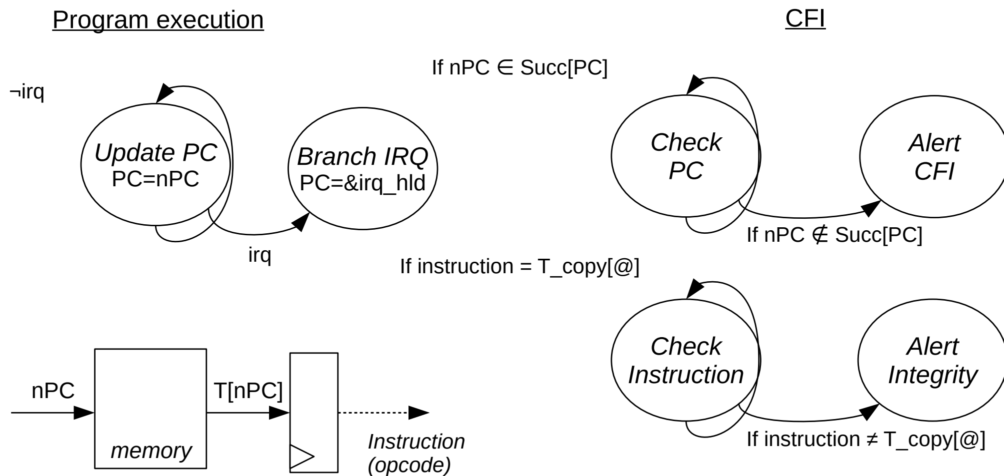


Figure 4.1: Program state machine (*left*) and CFI state machine (*right*)

Unfortunately, we have this fact:

Fact 1. *The verification of the good execution of P during the execution takes more time than executing the program himself. Indeed, executing one instruction and computing the next address is a single operation, whereas verifying the next address of the PC requires a search in a list.*

Actually, in Figure 4.1, the costly operations are those which require an indirection (Succ[.] and T_copy[.]), but the verification that a given vertex V belongs to a list can require more than one indirection (depending on the cardinality of the said list).

4.1.3 Basic-block based seamless verification of property P

We can improve Fact 1 by reducing the program's graph flow. This is achieved by merging some vertices together.

Definition 5. We define $\mathscr{R}$ a relation between two vertices V_1 and V_2 such that $V_1 \mathscr{R} V_2$ is verified if one of the following properties holds:

1. $V_1 = V_2$
2. V_2 only admits V_1 as a predecessor and V_1 only admits V_2 as a successor ($\text{Succ}[V_1] = \{V_2\}$ and $\text{Pred}[V_2] = \{V_1\}$)
3. V_1 only admits V_2 as a predecessor and V_2 only admits V_1 as a successor ($\text{Succ}[V_2] = \{V_1\}$ and $\text{Pred}[V_1] = \{V_2\}$)

Definition 6 (Equivalence relation)**.** We define $\mathscr{E}$ an equivalence relation between two vertices V_1 and V_2 as follows:

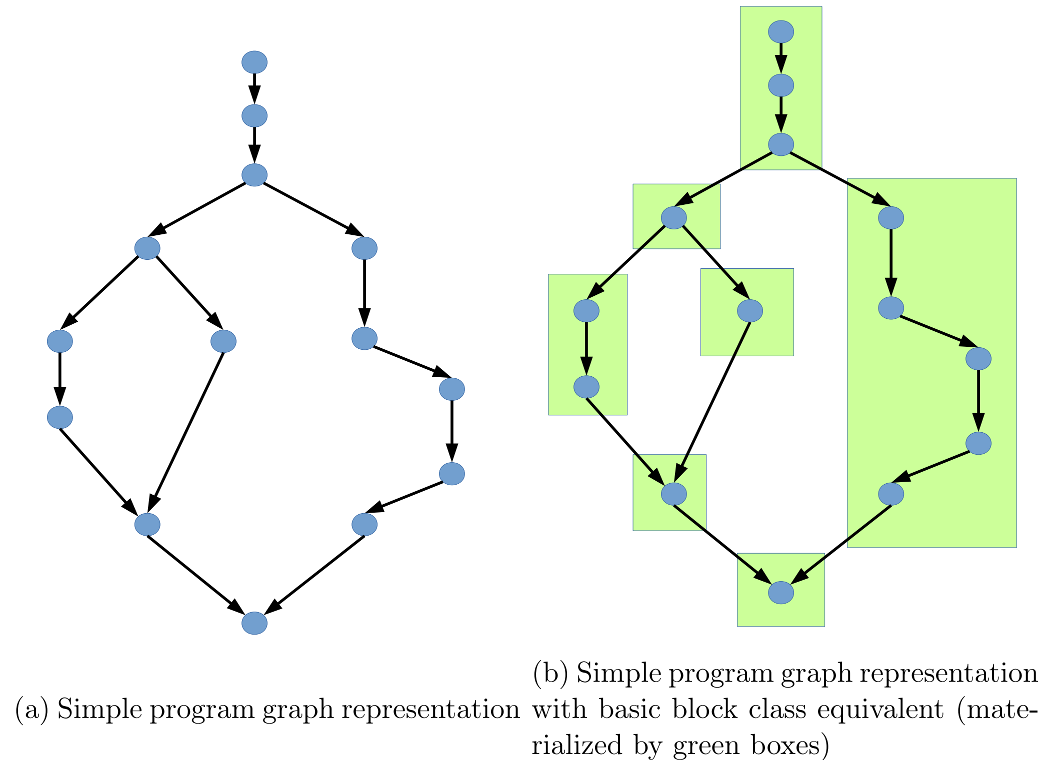
$$\begin{aligned} & V_1 \mathscr{E} V_2 \iff \\ & \exists n \in \mathbb{N}, \exists \{IV_i\}_{1 \leq i \leq n} \subset \{V, V \in \text{program flow graph}\} \\ & \quad \text{such that: } V_1 \;\mathscr{R}\; IV_1 \;\mathscr{R}\; IV_2 \;\mathscr{R}\; \ldots \;\mathscr{R}\; IV_n \;\mathscr{R}\; V_2 \;. \end{aligned}$$

A class of vertices that verify the equivalence relation $\mathscr{E}$ is called a *basic block*. A basic block contains at least one vertex V, in which case we have $V \mathscr{R} V$, and $n = 0$ in Definition 6 (i.e., there is no intermediate vertex). Applying the equivalence relation $\mathscr{E}$ regroups several vertices together, as depicted in figures 4.2a (*before simplification*) and 4.2b (*after simplification*).

We have this well-known result:

Proposition 1 (Control Flow Graph)**.** *A program graph can be uniquely partitioned into basic blocks.*

Proof. From the definition 6 it follows that the relation $\mathscr{E}$ is *reflexive*, *transitive* and *symmetric*. Therefore we can use $\mathscr{E}$ to define classes of equivalence to partition the graph of a program into basic blocks. □



(a) Simple program graph representation

(b) Simple program graph representation with basic block class equivalent (materialized by green boxes)

Figure 4.2: Simple program graph representation

This simplification of the *program flow* is called a *control flow graph* (CFG). It induces a simplification in the verification FSM, as shown in Figure 4.3.

With this simplification we are able to do the same security controls but with much less verification operations. Indeed, within a basic block, it suffices to check that the PC is incremented by one instruction. The rest is the verification of the jumps, which is referred to as CFI.

To implement basic block integrity verification, it is sufficient to compute a signature of the instruction sequence that has been executed. The notion of signature we use is the following:

Definition 7 (signature of a sequence of instructions)**.** A signature "Sign" is a function which can be computed incrementally, and which produces a result of the same size as the instruction, with the following property. Let

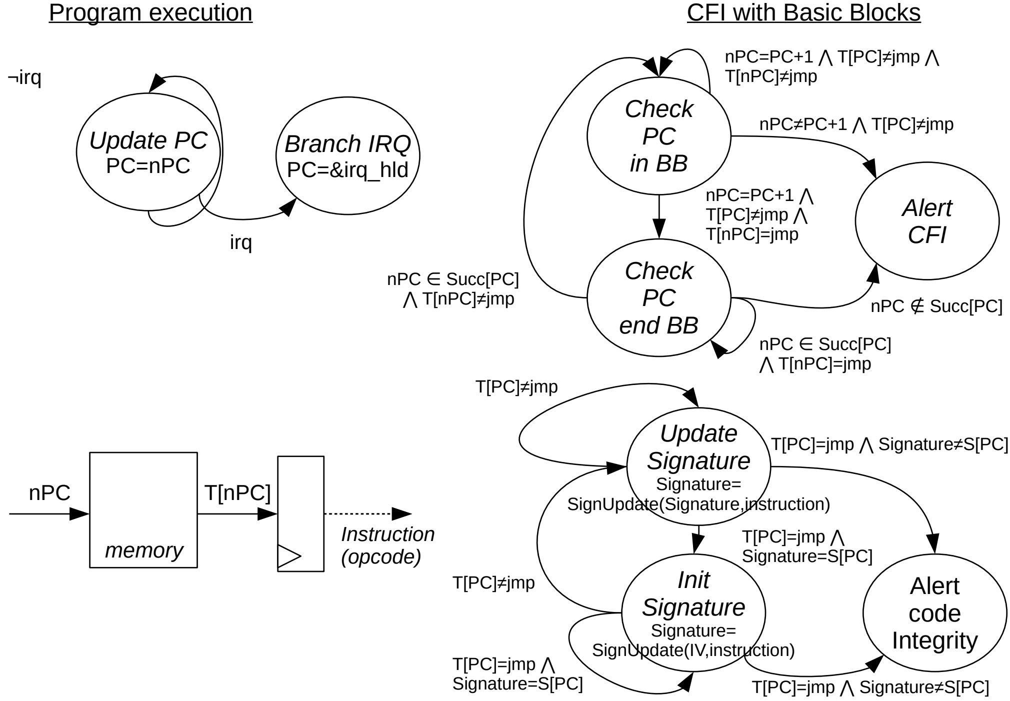


Figure 4.3: Program state machine (*left*) and CFI state machine (*right*), optimized to take into account basic blocks

$n, n' \in \mathbb{N}^\star$ (recall that a basic block is made up of at least one instruction, hence n and n' are taken nonzero). If

$$\begin{aligned}\text{Sign}(\text{instruction}_1, \ldots, \text{instruction}_n) &= \\ \text{Sign}(\text{instruction}'_1, \ldots, \text{instruction}'_{n'}) &\ ,\end{aligned}$$

then with very high probability, we have $n = n'$ and for all i $(1 \leq i \leq n)$,

$$\text{instruction}_i = \text{instruction}'_i \ .$$

In the sequel, we will implement the signature by concrete primitives, such as cyclic redundancy check (CRC). We might also refer to this CRC as “hashing”, although strictly speaking, we do not need the properties of a cryptographic hash function. The property of *anti-collision* presented in Definition 7 is very basic, and satisfied by usual CRC algorithms.

Remark 1. Actually, as represented in Figure 4.3, we make the classical assumption that signatures can be updated after each new instruction is processed. That is, Sign (in Def. 7) is made up of two functions:

1. one which initiates a signature with the first block (here, the first instruction), and
2. a second one which updates the signature by digesting further blocks.

The initial function is typically the update function (SignUpdate) with an initialization vector (IV) instead of the previous signature. And so, in Definition 7, the expression:

$$\mathrm{Sign}(\mathrm{instruction}_1, \ldots, \mathrm{instruction}_n)$$

is actually short for the following recursive expression:

$$\begin{aligned}
&\mathrm{Sign}(\mathrm{instruction}_1, \ldots, \mathrm{instruction}_n) = \\
&\qquad \mathrm{SignUpdate}(\mathrm{SignUpdate}(\ldots \\
&\quad \mathrm{SignUpdate}(\mathrm{SignUpdate}(IV, \mathrm{instruction}_1), \mathrm{instruction}_2), \\
&\qquad\qquad \ldots), \mathrm{instruction}_n) \ .
\end{aligned}$$

At this point, the only costly verifications on a graph are inter-basic block integrity checks. In practice, the FSM can delineate in real-time the basic blocks, since they follow a jump instruction (conditional test, function call, etc.) and end before a jump instruction. Then, it is sufficient to store the CFG and the signature of basic blocks instead of the whole binary code. We continue to refer to flow graph as Succ[@], but only those addresses @ that end basic blocks have a list of successor defined. Similarly, the binary code is replaced by the signatures of basic blocks.

Notation 2. *We also denote by S[@] the signature of each basic block, where @ is the representative address of a basic block, typically the last address (where the basic block integrity verification is done, as represented in Fig. 4.3).*

Therefore, the statically computed graph of all Succ[.] and of all signatures S[.] is drastically compacted with respect to the original ones (T_copy[.]

as described in Sec. 4.1.3). So, as illustrated in the bottom right-hand side of Figure 4.3, on jumps, the correct execution of instructions is checked by the correct signature of a basic block.

In addition, we have this remarkable result:

Theorem 1 (CFI+HCODE). *If both the CFI and HCODE are verified simultaneously during the execution of a program, then not only the CFG is unaltered, but also the* entire *program graph.*

Proof. The integrity of the program graph is checked:

- by the CFI on jumps, and
- by the HCODE signatures on exiting basic blocks.

Now, by Proposition 1, a program graph is completely described by its basic blocks and the CFG that connects them. □

Corollary 1. *When both CFI and HCODE are used to verify the two items of Property P (property 1), then it is useless to verify that the program counter addresses are incremented with basic blocks. This means that, in Fig. 4.3, the transition from state "Check PC in BB" to "Alert CFI" can be removed, since such assertion would be caught by the transition from state "Update Signature" to "Alert code integrity".*

Proof. This is a direct application of property of signatures given in Definition 7. If the program counter is not incremented linearly, then the sequence of instructions will change, and the signature will be erroneous, which will trigger an alert.

To be precise, the signature will be correct in the case where the instructions are replaced by exactly the same number of identical instructions. But in this case, it the program execution is correct, and there is no need to raise an alarm. □

In the following sections, we describe how we implement the mechanisms from Theorem 1 into the platform[1].

[1]For conciseness, we may denote in the sequel the verification presented in Theorem 1 as "HCODE" (instead of "CFI+HCODE").

4.2 HCODE Operating Principle

4.2.1 Big picture

In this section, we briefly summarize the rationale of HCODE real-time CFI.

The basic idea of the hardware support for control flow integrity checking is that a trace of the program counter is generated in real-time, and placed in a queue for verification purposes. Verification can be achieved concurrently, either by dedicated hardware, by software code running on another core, or sequentially by another process running on the same core. The hashing can be accelerated by a *batch* processing, possibly *out-of-order*.

Of course, as already highlighted in Fact 1, it is much more time-consuming to check that a sequence of program counter addresses travels correctly on a precomputed CFG than to execute a program. Indeed, most instructions are executed in a small (from one to a few units) amount of clock tics, whereas the verification that an address belongs to the list of possible successors of a vertex in a CFG can take a long time, depending on the complexity of the CFG. So, to relax the verification constraints, we have proposed in Theorem 1 a specific “shortcut” for checking basic blocks integrity. Instead of posting a series a consecutive addresses for verification, a hardware module computes a signature of the executed instructions. At the end of the basic block, the signature is checked against a precomputed one. The signature is specified to detect errors in case the previous (normally straightline) instruction sequence has not been executed as expected. So, the verification of a dynamically computed value against a precomputed one can be achieved in one clock cycle, which does not delay the CFG verification procedure. Furthermore, the signature verification module is idle during the execution of basic blocks (i.e., while the condition “T[PC] $\neq$ Jmp” is asserted in Fig. 4.3); it can thus take advantage of this time to perform the (more time-consuming) verifications of the CFG which have been delayed (e.g., queued in a FIFO).

4.2.2 Overview

HCODE is a hardware accelerator module placed between the processor and the instruction cache. It works by computing a signature of basic blocks

executed by the processor. A manual or automatic analysis of the assembler code is needed to get the control flow graph and precompute the signature of each basic block. They will be used during the execution of the program to verify the integrity of the executed instructions. During the execution of the program, the HCODE module intercepts each instruction fetched by the processor and calculates a signature (or a CRC, or a hash) until a jump instruction is fetched. Then HCODE module enqueues the CRC and the address relative to the basic block in one FIFO, and starts the computation of the new CRC for the next basic block. When the processor switches contexts, the operating system (OS) is responsible for emptying the FIFO and checking that each computed CRC present in the FIFO are the same as those present in the section '.HCODE' of the binaries (place where the precomputed CRC is stored).

A program with HCODE protection embeds one .HCODE section in its ELF object file. This new section contains all necessary information to do CFI and the correctness verification of executed instructions (.HCODE segment is explained in section 4.3.2). When the program is launched, the OS loads the program in memory as usual with all the additional data of the .HCODE segment in the process table.

The HCODE module can be disabled globally or locally depending of its configuration.

4.3 Software implementation

To benefit from the HCODE protection measure, we need to make little adjustments in the compilation toolchain. Indeed HCODE assumes that each basic block ends by a jump instruction, which is not necessarily the case if the next basic block is the target of an another jump instruction (see one example in Listing 4.1).

The figure 4.4 represents the modified compilation work flow for adding HCODE protection. The first step is to generate assembly code, in which we check whether each basic block ends with a jump instruction. If this is not the case an unconditional jump to the next basic block is added by taking into account the delay slots if necessary. In parallel we add a unique label of each start/end of basic block (see Listing 4.2).

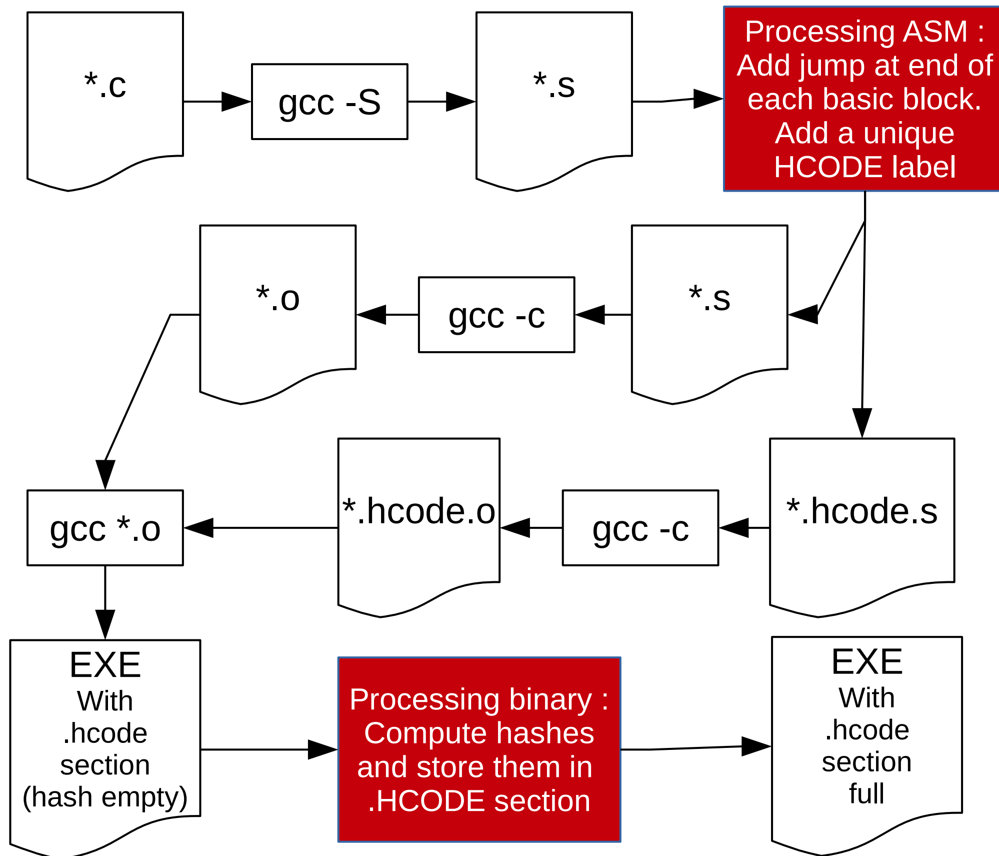


Figure 4.4: Modified compilation toolchain

4.3.1 Adding jump

```
1 lbl0:
2         mov  ...
3         add  ...
4 lbl1:
5         cmp  ...
6         jmp  lbl1
```

Listing 4.1: Basic block, which is not delineated by a “jump” instruction

```
.global hcode_start_0001
.global hcode_end_0001
.global hcode_start_0002
.global hcode_end_0002

lbl0:
hcode_start_0001:
          mov ...
          add ...
hcode_end_0001:
          ; jmp inserted by HCODE
          jmp hcode_start_0002
lbl1:
hcode_start_0002:
          cmp ...
hcode_end_0002:
          jmp lbl1
```

Listing 4.2: Basic block, separated by a *dummy* "jmp"

The second step is performed on the generated binary object file. For each portion of code we want to apply the HCODE protection to, we extract the basic blocks list and then compute the corresponding checksums. Finally, all needed pieces of information (list of successors and CRC) are stored in a new section of the ELF file we name '.HCODE'.

The program may not always be comprehensively protected by HCODE, as some part of the program can be dynamically loaded at the execution time (dynamic linked libraries) and therefore cannot be modified by the user. This justifies the functional requirement of being able to specify which parts of the program should be subject to HCODE protection.

4.3.2 The new '.HCODE' section

Once the ELF executable has been generated by the compilation toolchain, we must initialize the section '.HCODE' in the ELF file. After initialization,

this section contains all precomputed signatures of desired basic blocks. This section contains one or more entries. Each entry is defined by the end address of basic blocks, and contains all addresses of their successor along with the CRC of the current basic block instructions.

4.3.3 Hardware module

The HCODE hardware module is inserted between the instruction cache and the processor. It catches all opcodes fetched by the processor, the output of the program counter and the AMBA (Advanced Microcontroller Bus Architecture) bus, used by LEON (and most systems-on-chip, like ARM processors, etc.).

The HCODE module contains control logic, a signature (or CRC, or hash) module, a RAM on the AMBA bus used as a FIFO, and multiple config registers also on the AMBA bus used for internal configuration.

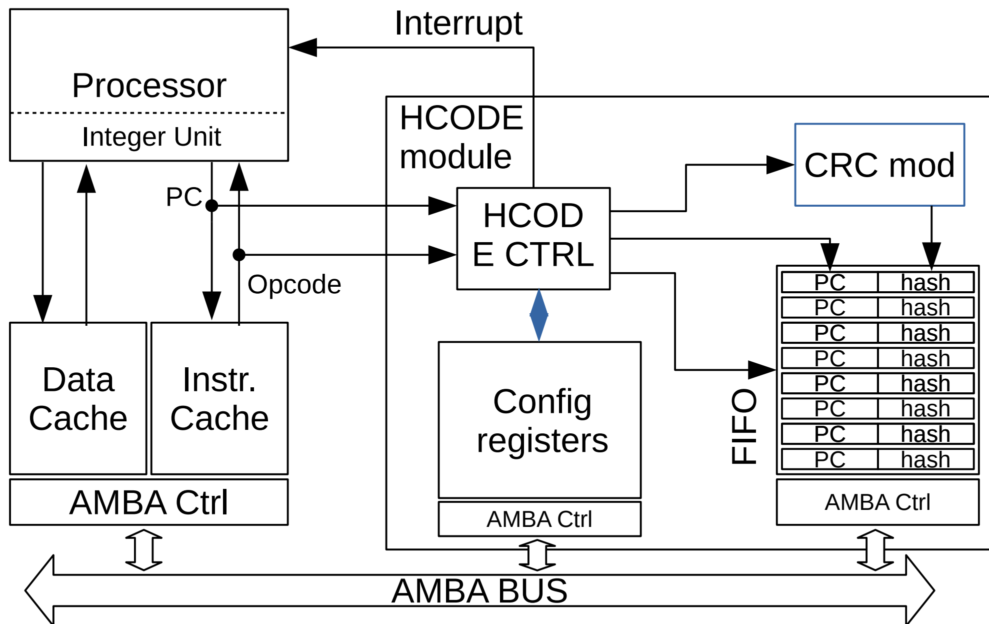


Figure 4.5: Hardware implementation (simplified)

At each processor cycle, the opcode of the currently executed instruction

is intercepted and used to update the current signature. If the module intercepts an opcode corresponding to a jump instruction, it updates the sum of control and posts it in a hardware FIFO along with the address of the current basic block of the program. After that, the hash module is reinitialized for the next basic block. Its FSM is more detailed in the section 4.3.3.

The HCODE module is addressable on the AMBA system bus and provides control registers:

- register CONF: this is the configuration register (On/Off; user/user and super mode; ...).
- register HASH: contains the signature of the current basic block.
- register PC: contains the record of the beginning of the current basic block.
- register NEXT: contains the next free line in the HCODE FIFO.

The HCODE module is designed to send an interruption when the FIFO (dual-port AMBA RAM) is full to prevent losing data. This causes a context switch forcing the OS to empty the FIFO and make all the necessary integrity verifications. This operation is more detailed in the OS section, namely section 4.3.5. Notice that, let apart this interruption, the HCODE module is totally passive with respect to the CPU: it can be inserted without any modification, since it only probes some buses (but does not modify them).

The verification of Property P is done outside of the HCODE module, by a code which validates the pairs (PC, hash) of the FIFO against the precomputed data stored in the .HCODE section in the binary.

HCODE FSM

The HCODE has two states: active and inactive, depending on the CONFIG register, user or super mode and also page memory settings.

In the inactive state, opcodes are just forwarded to the processor, and no hashing is done. It prevents the FIFO from getting full too quickly. Inactive states are used by the kernel when it is running processes without support

for HCODE. In this case, context switches do not go through the verification process and "HCODE MEMORY FULL" interruptions are not triggered.

In the active state, every opcode is hashed depending of the current basic block. In the hardware implementation, availability of a new opcode is indicated by the 'op' signal, '¬op' corresponding to a processor stall. Every address of the beginning of a basic block is saved in a register (PC). At the end of a basic block – indicated by the 'end_BB' signal, the hash is pushed in a FIFO, along with the saved address of the basic block.

The active mode is described by the FSM depicted in Figure 4.6.

4.3.4 LEON 3

For implementation and experimentation purposes, the LEON3 processor is chosen. It is a compact processor implementing the SPARC v8 instruction set including multiply and divide instructions and a 7-stage pipeline. Our configuration includes a debug support unit (DSU), a 32×32 multiplier, data and instruction caches with MMU and 8 register windows. The LEON3 is integrated on a Virtex 5 FPGA (xc5vlx110) on the Gaisler board `GR-PCI-XC5V`. The prototype is shown in figure 4.7. The figure 4.8 summarizes the architecture of the processor, as provided by Gaisler.

Although the examples are carried out with a LEON3, they can be backported to almost every processor. As previously mentioned, the HCODE module is a "blackbox" (i.e., it can be plugged without altering the nominal behavior of the process), placed between the instruction cache and the integer unit pipeline (see Figure 4.9). Since we do not modify the binary code of the executed program, we just wiretap the requested instruction.

To be as general as possible, we assume that only the program counter and the opcodes are fed into the blackbox from the integer unit.

In order to raise interruptions, we also use the interruption line passing through the module.

4.3.5 Software OS

HCODE verification

In a firmware environment, the HCODE module triggers the "HCODE MEMORY FULL" interruption which is used to signal to the firmware that the FIFO memory of the HCODE is full, and that an action from the software is required. In a kernel context, every context change is used to perform all the verifications and flush the HCODE memory. But the interruption is also used if the program has a high priority, and in this case, the memory could overflow.

The action required to flush and process the HCODE memory is as follows:

- Dump the FIFO RAM from the AMBA bus;
- For each line in the RAM, check that:
  - the current PC of the line is in the list of accepted addresses from the previous line. If it is the first line, it uses the saved "last address" or the start address of the program if it is the first time that the verification is done;
  - the linear block has the same hash as the precomputed one.

For optimization reasons, the processing of each line can be *out-of-order*.

Since the hardware module does not check whether a basic block is included in the .HCODE section, the kernel or the firmware ignores every line in the FIFO corresponding to a non-protected basic block.

Kernel context

When the kernel starts a new process, it allocates a new `struct` to save information about the new process. In the case of a program with HCODE support, we modify this `struct` to save the control flow graph (CFG) and the HCODE blocks inside the `struct`. We also reserve an entry in the `struct` to save the internal state of hash module and the first address for the CFG inspector.

To handle context switching in kernel-land, the following pseudo-code are added:

- Do the HCODE verification (see Sec. 4.3.5);
- Dump the configuration RAM via the AMBA bus. This dump contains:
  - the internal state of the hashing module,
  - the saved program counter corresponding of the begin of the basic block,
  - the address of the next free line in the HCODE memory.
- Save in the configuration and the last line of the HCODE memory in the process `struct`;
- Write back the configuration RAM of the HCODE with the previously saved configuration. If it is a new process, use a blank configuration;
- Write back the HCODE block in the first position in the HCODE memory;
- Reset the next address of the HCODE memory using the configuration RAM and make it point to the first line;
- Continue normal context switch operation in the kernel.

4.4 Performance estimation

4.4.1 Hardware performance

Since all transitions in the HCODE FSM are done in one cycle, there is no latency in the treatment of an opcode.

The sizeable part of the HCODE module is the FIFO RAM whose depth is adapted to reduce unwanted context switches due to the "HCODE MEMORY FULL" interruption. In our programs, the mean size of a basic block is ten instructions. So providing an approximation of the number of instruction per context-switch, it is possible to eliminate unwanted context switches.

4.4.2 Software performance

Since we are not introducing many additional instructions in the original program unlike other software based control flow integrity implementations, the performance cost in greatly reduced.

Our embedded firmware (an AES [52] test code) composed of 2545 instructions, only 81 jumps are added to permit the HCODE module noticing the end of a basic block. Dynamically, for each AES execution, 9220 jumps are executed before modification, and 10934 after.

In most cases, the overhead is small (like in our AES test code), but in case of program with a lot of added-jumps (program with lots of loops), this performance impact could be mitigated by the addition of a new instruction closing the basic block. This one cycle instruction should not update the hash but permit to save and init again the hash module.

So in the user mode, the performance is globally the same as that of an unmodified program.

The main performance impact is due to the context switch since this is when all the verifications are made.

In the GNU/Linux OS, the context switch is very expensive. Dumping and checking a FIFO is not the most expensive part of the switch code.

4.5 Conclusion

In this chapter we present the basis of functioning of our solution. We introduce the CFI verification and the inter-Basic-Block verification called HCODE. In this chapter demonstrate that is possible to ensure a full fine-grain CFI with CI with hardware module without modifying the processor. we have seen that with little additional hardware, without (or nearly) adding code instruction, we are able to significantly mitigate the corruption of the instructions flow, without much slowing down the user process.

Nonetheless, the strategy of adding a jump at the end of each basic bloc to detect the end of it is costly at the runtime. We also have highlight that research of the signature in the .HCODE section is time consuming. In the

next chapter we present a new approach using metadata to store needed information such as the signature to overcome these limitations. We also present a new innovative approach using hardware cache to speed up the metadata fetch during the execution time.

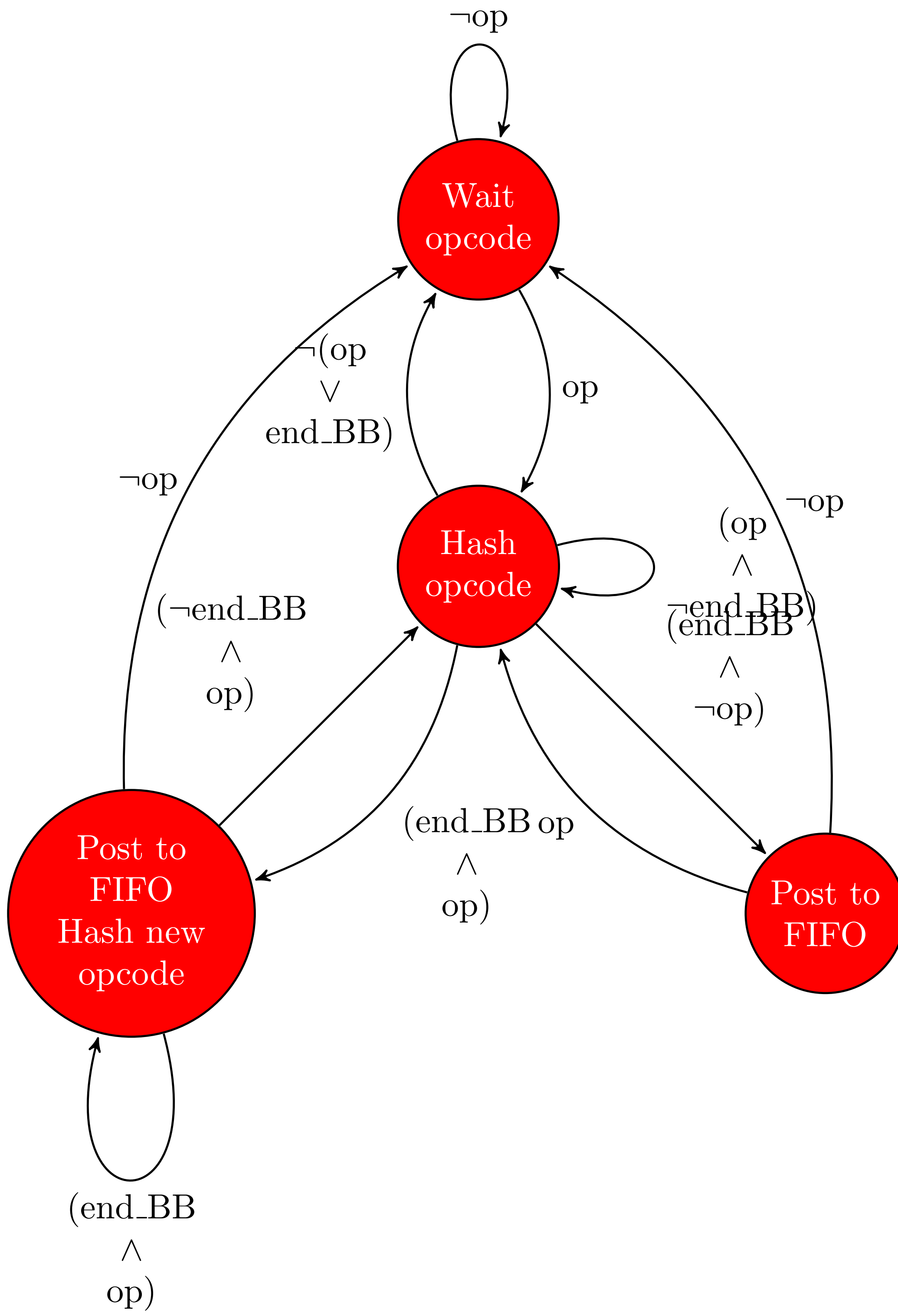


Figure 4.6: FSM of the HCODE controller hardware module

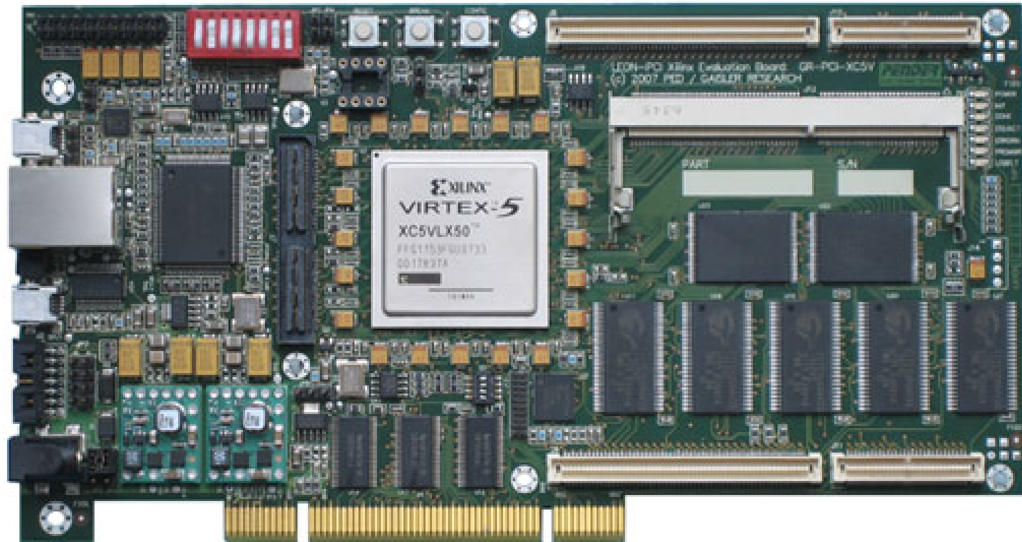


Figure 4.7: Gaisler board used for the LEON3 prototype

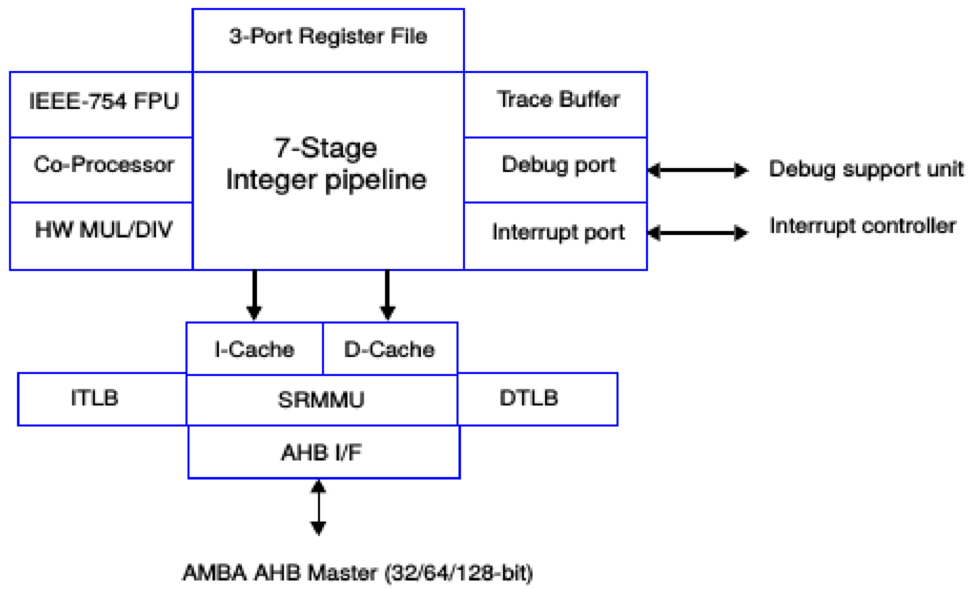


Figure 4.8: LEON 3 internal

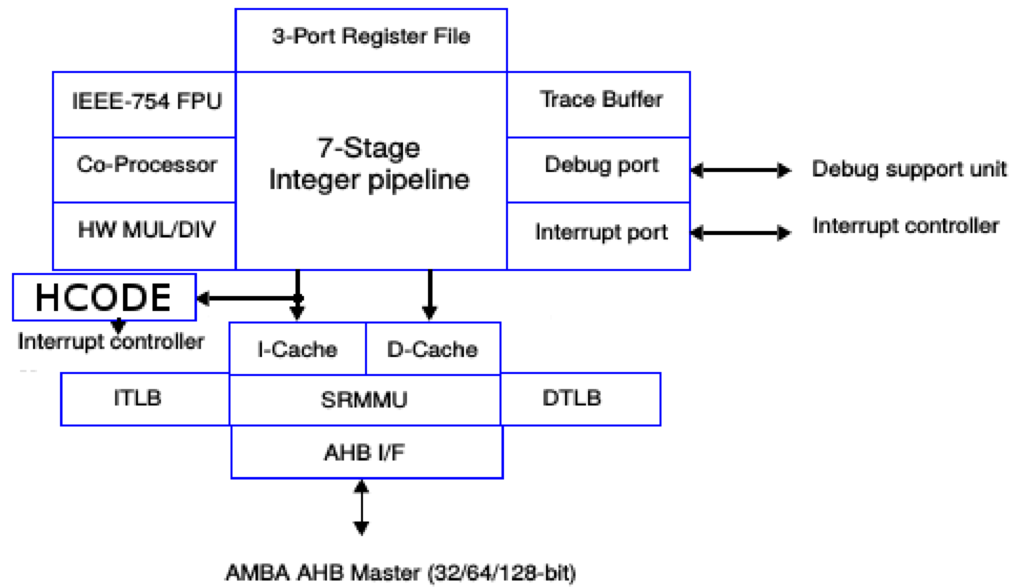


Figure 4.9: LEON 3 internal with HCODE module

Chapter 5

CCFI: Code and Control Flow Integrity

5.1 Solution

In this section, we present the principles of our proposed solution, before discussing implementation issues in Section 5.2.

5.1.1 Architecture Overview

The basic architecture is shown in Figure 5.1. We consider a simple platform based on a CPU core with separate instruction and data cache, which connect to the memory bus. CFI is ensured by two added hardware modules (shown in red): The *CCFI-cache* fetches the metadata which has been computed at compile-time, containing all control-flow related information. This information is used at runtime by the second module, the *CCFI-checker*. In order to follow and monitor the execution of the CPU, the CCFI-checker is hooked up to the interface signals between the CPU and the instruction cache.

The CCFI-cache has the same characteristics (bit width, size, associativity, replacement policy, ...) as the instruction cache. For each basic block in the executed program, there is a corresponding block of metadata. Each block of metadata is perfectly aligned in memory to its corresponding BB,

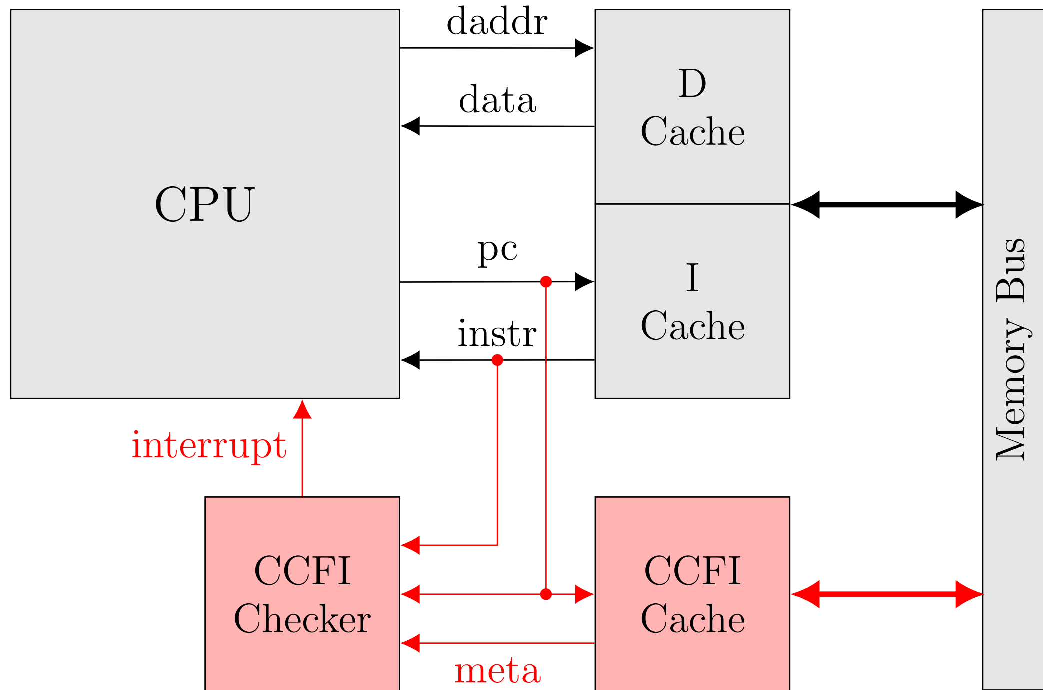


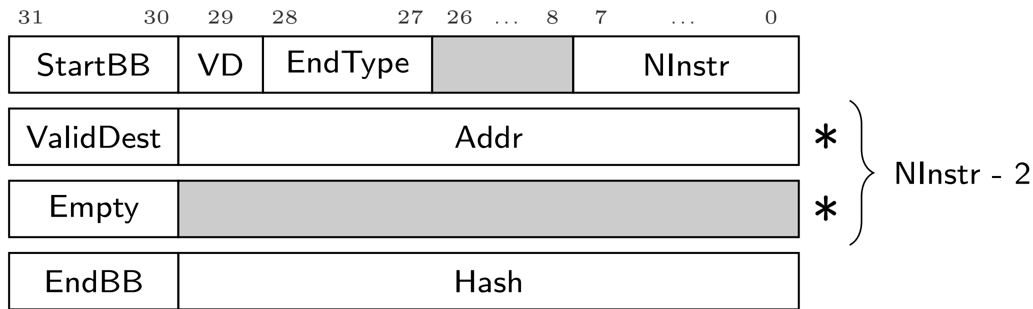
Figure 5.1: Overview of the proposed architecture

with a constant offset. For each access to the instruction cache, a parallel access to the CCFI-cache will be issued. In this way, complex address calculations are avoided. Furthermore, the instruction cache and the CCFI-cache will always be consistent, i.e. either both a BB and its metadata are cached or none of them.

The metadata for each BB contain three crucial elements that serve for the CFI verification:

1. The number of instructions in the current BB
2. The valid destination addresses of the succeeding BBs
3. A hash value of the instructions in the BB and its corresponding metadata

Section 5.1.2 discusses in more detail the format of the metadata.



EndType ::= Branch | Call | Return

Figure 5.2: Metadata format description

The actual verification is realized by the CCFI-checker. At the end of each BB, it checks the validity of the target address by comparing it with the precomputed valid addresses contained in the metadata, thereby ensuring intra-procedural CFI. In case of a function call or return, an integrated shadow stack is used to verify inter-procedural CFI. This shadow stack in embedded inside de CCFI-Checker and is not accessible from the main processor. Intra-BB consistency is ensured by a watch-dog counter that controls the number of executed instructions before a control transfer. Finally, code and metadata integrity is ensured by a precomputed signature that is compared to a hash value over the executed instructions computed at run-time. In case of any violation, an interrupt is raised. The details of the CCFI-checker are presented in Section 5.1.3.

5.1.2 Metadata

The format of the metadata is shown in Figure 5.2. There are four different entry types, which are distinguished by a label contained in the two most significant bits. For each BB, the metadata record has the same format: A StartBB entry, followed by zero or more ValidDest and Empty entries, and ending with a EndBB entry.

The entry StartBB marks the beginning of a BB. The bit VD indicates the presence of one or more valid destination addresses in the record. Note that in some cases destination of indirect branch or jump cannot be computed

statically, in this case the VD bit is unset and there will be no verification of the destination address at the end of the BB. The field EndType defines the type of control transfer at the end of the BB: Call and Return indicate a function call and return, respectively. For any other type of control transfer, Branch marks either a BB that will always be succeeded by the next consecutive BB – i.e. there is no branch or jump at the end – or one ending with a direct or indirect jump or branch instruction. Typically, blocks ending with a direct jump or call will have one valid destination and conditional branches two, while indirect control transfers can have an arbitrary number of valid destinations. Finally, there is an 8-bit field NInstr, which gives the total number of instructions in the BB.

The ValidDest entry contains one valid destination address, corresponding to an allowed edge in the CFG. Finally, the end of the BB is marked by an EndBB entry, which additionally contains a hash signature computed over all the instructions of the BB.

All metadata are computed offline at the end of the compilation, after code optimization. For each BB a metadata record of the same size is allocated. Metadata are stored in a custom section of the program file, which has the same size as the `.text` section.

If the number of entries needed for the CFI information is smaller than the number of instructions in the corresponding BB, then the metadata is simply padded with Empty entries before the EndBB entry. The opposite case can occur as well, if the BB is very short or if there are multiple valid address entries. In order to match the BB size with the metadata record, the compiler inserts `nop` (no operation) instructions just before the last instruction of the BB.

Depending on the specific implementation of the CPU, there can be other situations that require adjustment of the binary code. One such case is branch prediction, which potentially leads to a mismatch between the *fetched* instructions and those that are effectively *executed*. Since the CFI-cache only monitors the interface signals of the CPU, it needs to be aware of such features in order to correctly detect the destination of branches. Section 5.2 explains how we have resolved this issue for the used RISC-V implementation and gives an example for the correspondence between binary code and metadata. We evaluate the performance penalty of these code adjustments in Section 5.3.

5.1.3 CFI Verification

At loading time, the `.metadata` section is loaded into a reserved memory region, at a constant offset from the `.text` section. This offset allows calculating the metadata address on-the-fly based on the current instruction address.

During the execution of the program, the instruction cache and the CCFI-cache are always kept in a consistent state, which allows the CCFI-checker to follow the execution and verify the control flow on-the-fly using the metadata. The checker processes the metadata in parallel to the execution. For this purpose, the CCFI-checker is composed of the following principal components:

- A set of registers to store the valid destination addresses,
- A shadow stack to store function return addresses,
- An instruction counter, and
- A signature register to compute a hash value of the executed instructions.

A simplified view of the control state machine of the checker is shown in Figure 5.3. The state machine basically follows the structure of the metadata record (cf Figure 5.2). At the beginning of a BB (state **Start BB** in Figure 5.3), it sets the instruction counter to the number of instructions in the BB and initializes the signature register. It also checks that the beginning of the BB is correctly labeled with StartBB. The valid destination addresses are collected while traversing the BB (state **Store Dest**) and stored in the internal register bank[1]. If there are Empty entries in the metadata record, the state machine loops in the **Inside BB** state until the end of the BB. During the traversal, the signature register is updated after each instruction. For this purpose, a suitable hash digest function H needs to be chosen.

The actual verification takes place in the **End BB** state. There are two conditions that each triggers the transition into this state: Either an EndBB

[1]Note that the size $k \geq 2$ of this register bank is an implementation parameter that can be chosen freely. Any BB with more than k valid targets can be split recursively until each BB has at most k valid successors.

label is found or the instruction counter reaches zero. This ensures that both too long and too short BBs will be detected immediately. Normally, the end of the BB should coincide with the instruction counter reaching zero, which is verified in the **End BB** state. It is also checked if the hash value extracted from the EndBB entry equals the signature register. Call and Return are verified using the shadow stack. If there have been any valid address entries in the metadata, these are used to verify the effective target address. Note that this applies either for calls or local branches and jumps. The implementation of the internal register bank must ensure that the address comparison can be performed in parallel for all valid entries in one clock cycle, before the state machine continues to process the next BB.

In case any of the checks fails, an interrupt will be triggered, allowing the CPU to react to the attack immediately. Note that for simplicity reasons, Figure 5.3 does not show the state transitions in the case of a security violation.

5.1.4 Attack Model and Security Guaranties

In this work, we address the protection of embedded platforms without DEP. We consider that the attacker is able to exploit programming bugs which allow buffer overflows. Such attacks can either modify the return address on the stack and/or inject malicious code. Note that this attack model is quite permissive in contrast to the classical CFI setting, which usually considers that code memory is immutable [4].

Additionally, we consider non-destructive physical attacks. This includes random changes in memory by either software-driven attacks (row-hammer) or hardware attacks (such as electromagnetic injection or glitches) leading to instruction skips. Since the successful demonstration of practical attacks such as row-hammer, physical attacks must be considered a realistic scenario, especially in the context of embedded and mobile devices.

Assuming that the main memory contains the code alongside with its correctly generated metadata, the CCFI-checker enables detection of the following attacks:

- Changing a return address on the stack (detected by shadow stack)

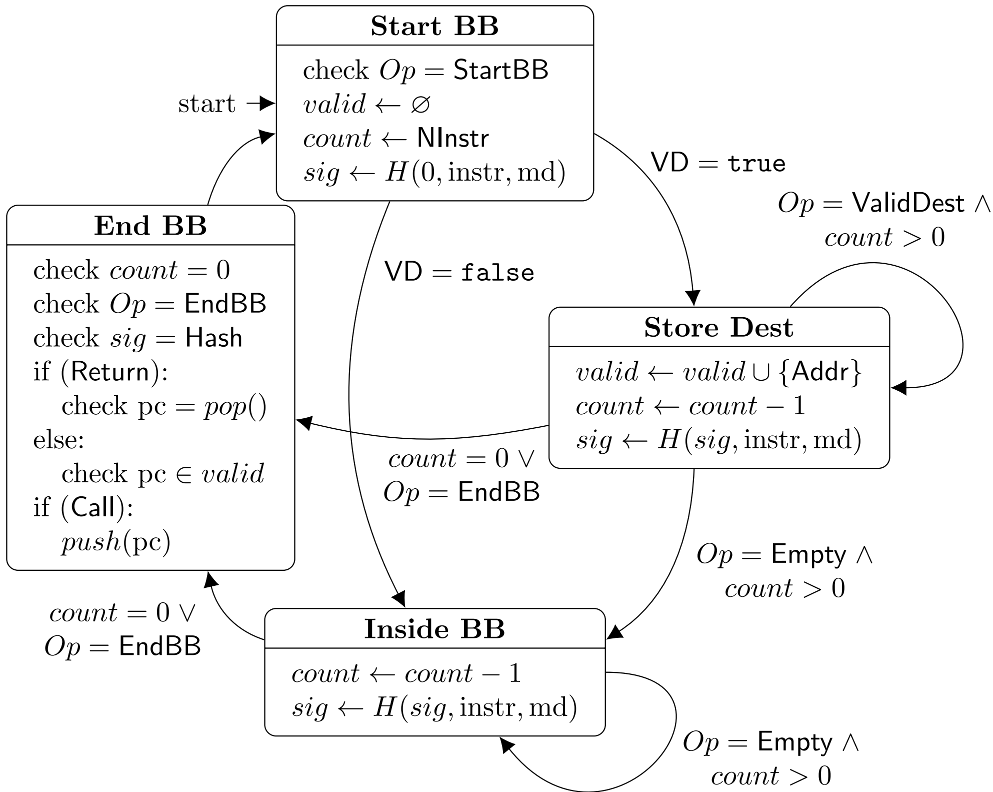


Figure 5.3: CCFI-checker state machine

- Changing the target of a call, branch or jump outside of the static CFG (detected by destination address verification)
- Returning or jumping into the middle of a BB (detected by StartBB label check)
- Adding instructions at the end of a BB (detected by EndBB label check and instruction counter)
- Turning a branch into a `nop` (detected by signature check)
- Changing the pc to skip an instruction (detected by signature and instruction counter)
- Changing any instruction word in memory or up to the CPU interface (detected by signature)
- Deleting or manipulation metadata in any way inconsistent with the code (detected by signature)

One obvious limitation are physical attacks that directly affect the internal state of the CPU, such as the register state or skipping computations within the pipeline. Note that however such attacks will be caught if they directly or indirectly change the instruction address on the cache interface. We also do not consider advanced destructive attacks (such as focused ion beams) that could e.g. cut the interrupt line on the circuit die and thereby practically disable the CCFI-checker.

Furthermore, data only attacks that do not change the static control flow are not detected. An attacker that has full control over the memory could also forge metadata. A typical solution for this problem is to assume that the metadata (or the metadata and the code) reside in a protected read-only memory. Considering our proposed architecture in Figure 5.1, such a solution can easily be implemented by having a completely separated memory bus for the CCFI-cache, thereby preventing any access to the metadata originating from the CPU.

Finally, special care needs to be taken for the treatment of the interrupt triggered by the CCFI-checker. Since interrupt mechanisms vary greatly on different target platforms, there is no system-independent solution. For instance, if interrupt target vectors are writable from user code, the interrupt

service routine (ISR) itself needs to be protected. Any tampering with the ISR would then lead to a re-occurring violation, blocking the system in an infinite loop. In embedded platforms, watchdog counters are typically used to reset the system when it gets caught in a deadlock. Depending on the application, if recovery is considered less important, the interrupt line of the CCFI-checker can also be routed directly to the reset line, preventing any attack path via the ISR.

5.2 Implementation

To validate the CCFI architecture, we have implemented it on a microcontroller platform based on the PicoRV32 [74], a free implementation of the RISC-V ISA [11]. The CPU core is a 3-stage pipeline processor with a single memory interface for accessing instructions and data. In our platform, the separate instruction and data caches are accessed via an address decoder. The platform uses a crossbar for memory access. To highlight memory contention between instruction cache and CCFI-cache, we have implemented two different memory layouts: 1) Two separate memories for application code (`.text` and `.rodata`) and metadata, 2) a single memory to store both. In all cases, our platform uses one big RAM as runtime memory.

As mentioned in Section 5.1.2, branch prediction can potentially pose a problem for the CCFI-checker, since the address seen on the memory interface (i.e. the program counter) may not coincide with the effective branch target address. The PicoRV32 does not feature branch prediction, but branches are decided late in the pipeline, such that the instruction following a conditional branch is always fetched. We resolve this specific case during the compilation phase by always inserting a `nop` instruction after a conditional branch, thereby deferring the actual control transfer by one instruction. We claim that the proposed architecture is suitable for more complex prediction schemes, for instance using a checker that mimics the prediction logic. A detailed discussion of the required modifications is beyond the scope of this paper.

Table 5.1 shows an example with two BBs and the corresponding metadata. The upper BB ends with a conditional branch and there are two valid destinations stored in the metadata record. In the code, there are two addi-

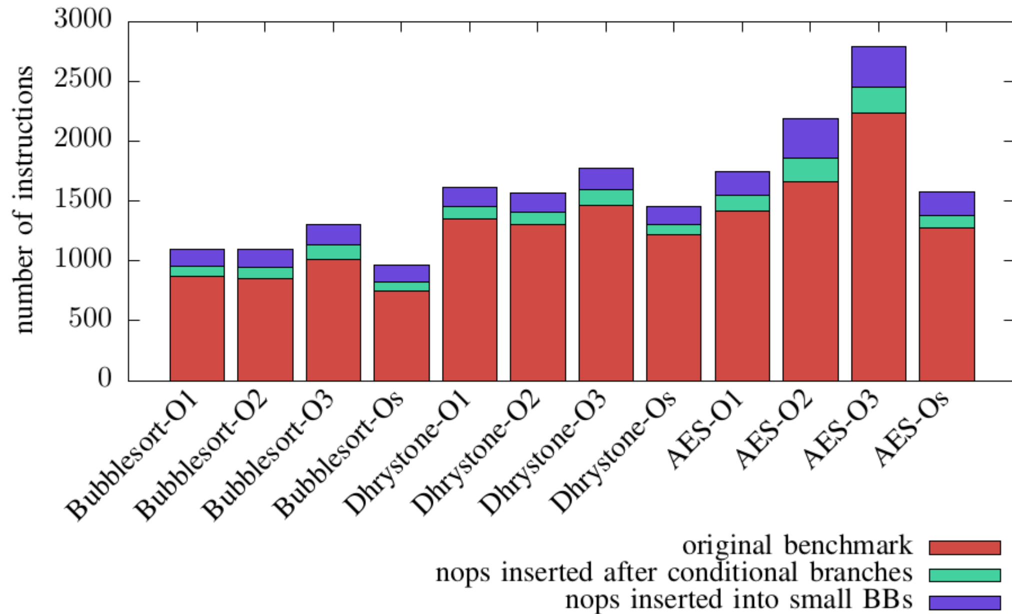


Figure 5.4: Overhead on code size

tional `nops`, which are needed to match the size of the metadata, and – for the second one – to resolve the prefetch of the conditional branch.

5.3 Performance

We have tested the solution on a Digilent Nexys4 DDR board with an Artix 7 FPGA [30]. The resource usage is summarized in Table 5.2. As can be seen, the hardware overhead is small, it is in the order of 10% in terms of LUTs and FFs [2].

Figure 5.4 shows the impact of the inserted `nop` instructions on the code size. Over the different benchmarks (bubble sort, Dhrystone, and an AES encryption) and optimization levels (`-O1`, `-O2`, `-O3`, and `-Os`), the overhead ranges from 9% to 30%.

[2]The difference between total CCFI and the sum of CCFI-cache and checker is due to glue logic and the more complex memory and bus infrastructure.

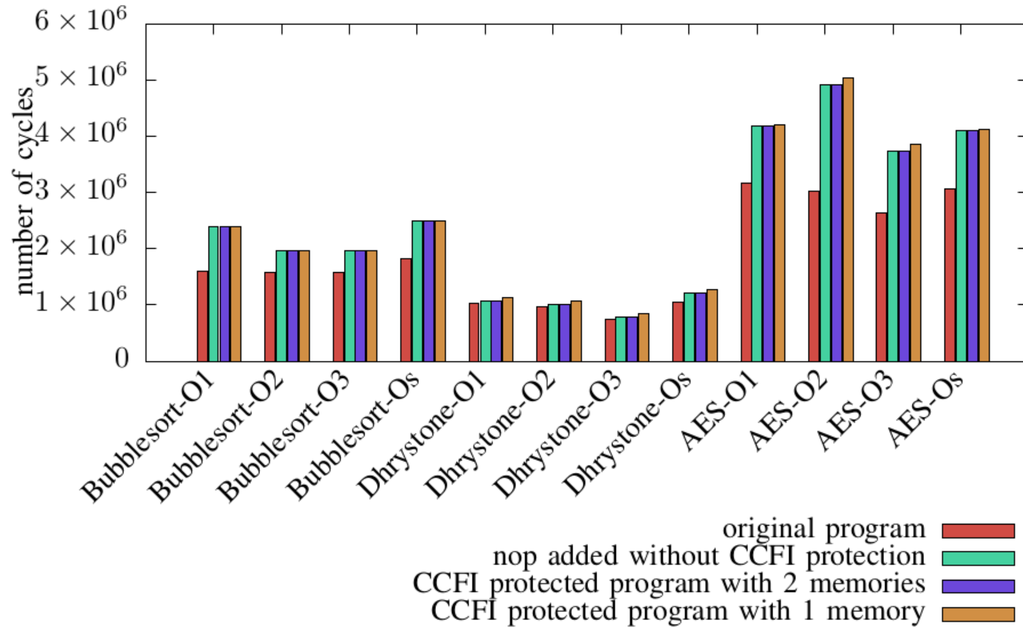


Figure 5.5: Overhead on execution time

Figure 5.5 shows the impact of our solution on runtime performance. We compare 1) the original program with 2) the modified program, but without protection, 3) CCFI enabled with parallel memory access, and 4) CCFI enabled with sequential memory access. We can see that most of the performance penalty is due to the inserted `nop` instructions leading to a runtime overhead between 2% and 63%. Beyond this overhead, the CCFI protection using parallel memory access does not further impact the performance. The implementation using sequential memory access incurs a small additional cost in the order of 1% (up to 8% in the worst case), which is directly related to the instruction miss rate of the benchmark.

Fault attacks. We have simulated fault attacks by modifying an instruction code in the main memory and in the cache. All performed attacks have been detected by the CCFI-checker, through the signature check. Any fault injection inside the processor which directly manipulates the program counter is detected as well.

Software attacks. We have also tried several software attacks on the protected platform. Any buffer overflow manipulating the stack is detected thanks to the shadow stack. Even changing *unprotected* indirect jumps is detected if the destination address is not the beginning of a BB (labeled by a StartBB entry). Thus, ROP or JOP attacks are made very difficult, since the number of useful gadgets is significantly reduced.

5.4 Conclusion

We have presented a non-intrusive hardware-based protection able to effectively mitigate cyber- and physical attacks. Our solution uses precomputed control flow information which are verified at runtime. Only requiring a double code memory size, our solution is very competitive regarding the hardware overhead and the performance penalty, which are minimal and affordable in most cases which make our technology practical and deployable.

Up to now, the metadata are computed as a post-processing step using the binary code. The integration of this step into the compiler flow is left to future work, to have better control over well sized BBs and the resolution of indirect jump destinations.

Table 5.1: Code and metadata correspondence

Instruction address	Instruction	Metadata address	Metadata	Metadata description
0x00000A88	`lbu a5,0(s1)`	0x40000A88	0xA0000004	StartBB | VD=1 | EndType=Branch | NInstr=4
0x00000A8C	`nop`	0x40000A8C	0x4000029A	ValidDest | Addr=0xA68
0x00000A90	`bnez a5,0xA68`	0x40000A90	0x400002A6	ValidDest | Addr=0xA98
0x00000A94	`nop`	0x40000A94	0xFC035B60	EndBB | Hash=0xFC035B60
0x00000A98	`lw a5,-68(s0)`	0x40000A98	0xA0000004	StartBB | VD=1 | EndType=Branch | NInstr=4
0x00000A9C	`addi a5,a5,1`	0x40000A9C	0x40000118	ValidDest | Addr=0x460
0x00000AA0	`sw a5,-68(s0)`	0x40000AA0	0x0	Empty
0x00000AA4	`j 0x460`	0x40000AA4	0xDAF87E5C	EndBB | Hash=0xDAF87E5C

Table 5.2: Hardware cost

Component	LUTs		FFs		RAMB18	
Original	11094		8939		8	
CCFI-cache	531	(+4.8%)	139	(+1.6%)	8	(+100.0%)
CCFI-checker	294	(+2.6%)	443	(+4.9%)	0	(+0%)
Total CCFI[2]	1250	(+11.3%)	777	(+8.7%)	8	(+100.0%)

Chapter 6

Interruption and Speculative Execution

In previous chapters, we present the basic operation of the CCFI solution, in this chapter we present modification apported to the solution to be able to protect interruption and processeur using out-of-order execution. In fact interruption can divert the control flow of an program at anytime to run an interrupt routine. From an external point of view this can be seen as an abnormal processor behaviour. We present a modification to the metadata to ensure that interruptions are detected and protected appropriately. Branch prediction and speculative execution are also introduce difficulties to our solution. Previous exposed solution rely on the fact that intrcution fetch from the processor to the cache is representative of the execution flow. This assumption is no longer correct when processor use complex mechanism to fill up its pipeline. We present is this chapter changes made to ensure full CFI protection to processor using these technique.

6.1 Solution

In this section we present changes bring to the architecture and to the metadata of our solution. First we present the hardware modifications part by presenting the integration of the CCFI with an processor then detail the internal architecture of CCFI parts. In a second section we present changes

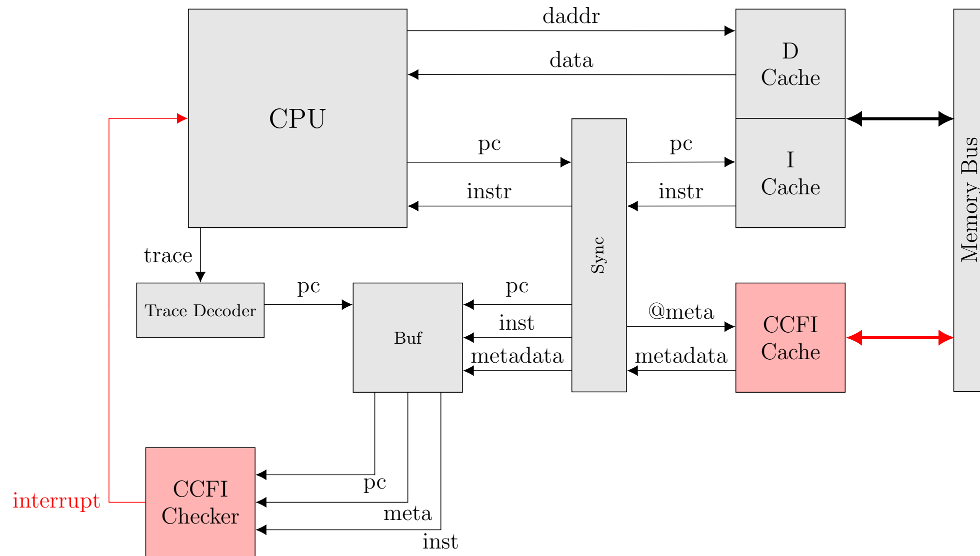


Figure 6.1: Overview of the proposed architecture

bring to the software part, metadata structure and toolchain modification to generate them. Section 6.2 present specificity needed in the architecture to be to protect processor with speculative execution. Section 6.3 present the modification introduced in the *CCFI-checker* and metadata to handle and protect interruptions.

Our goal is to provide the same level of security than the previous exposed solution on more advanced processor processor, namely CFI at Inter Pro **Inter Procedural**, **Intra Procedural** and **Intra BB** levels.

6.1.1 Hardware

The platform architecture of the solution is shown in Figure 6.1. We consider a simple platform based on a CPU core with separate instruction and data cache. CCFI is divided in five different module, two ensuring the CFI and CI: *CCFI-Checker* and *CCFI-Cache* (in red in figure 6.1). The other three are adaptation module which allow to adapt the solution to many different core.

The *CCFI-cache* fetches metadata which has been computed at compile-

time, containing all control-flow related information. This information is used at runtime by the second module, the *CCFI-checker*. The *CCFI-cache* has the same characteristics (bit width, size, associativity, replacement policy, ...) as the instruction cache. *Sync* module on the figure 6.1 emit the correct @meta given pc from CPU. It also act as barrier to synchronise the response of the instruction cache and the *CCFI-Cache*. When instruction and metadata are ready it sends both plus the actual PC of the request into *Buf* module which is implemented as a circular buffer. The purpose of *Buf* is explained in details in section 6.2. The purpose of *Trace Decoder* module connected to the trace interface of the processor is to extract the PC of the current instruction executed. The need of this module come from the fact that trace interface is not standardized. Some processor give all information such as address of the instruction and instruction itself. Other give less information like if a conditional jump is taken or not. This is why for some processor we need the *Trace Decoder* which will compute or extract the PC from the trace interface. Once PC extracted, it is send to *Buf* which given PC return data stored (instruction and metadata) of the given address to the next module, *CCFI-checker*. The actual verification is realized by the second hardware module *CCFI-checker* like previously. For each instruction issued from the trace interface *CCFI-Checker* receive at the same time the corresponding metadata of the current instruction, send by by the *Buf*.

6.1.2 Software

Metadata

Metadata contains informations depicting the Control Flow Graph of the program. These are organised in individual blocks of metadata each depicting the behavior of the execution of one Basic Block. Figure 6.2 show which informations are stored in these block of metadata and how it is formatted in memory for a 32bits architecture.

These metadata contain for each basic block the list of valid destination accessible at the end of the BB. It also contains a signature of the basic block computed from its instructions as well as the number of instruction present in the BB and other informations depicted in this section. These metadata have the same organisation as presented in section 5.1.2.

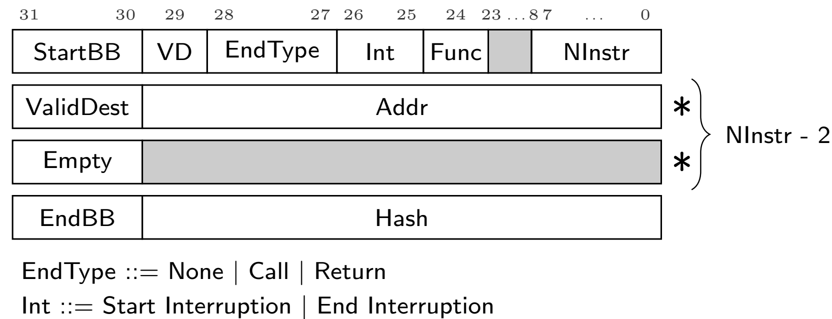


Figure 6.2: Metadata format description

Two field have been added to the StartBB entry. The first one is a 1 bit flag Func that was added to mark the first BB of functions. In the previous implementation, when a BB end by an indirect `call` that we don't know the allowed destination, the only restriction was the destination must be the beginning of a BB, same as for an indirect `jump`. This modification allow to restrict destination this type of `call` only to the beginning of a function. The second field is a 2 bits width Int, allowing to mark the beginning and the ending of an interruption routine. This flag allow the `CCFI-cheker` to detect on-the-fly when a interruption occur. If the execution flow divert from the CFG, `CCFI-checker` will check this flag before raising an alarm. If this flag is set to mark the beginning of an interruption routine `CCFI-checker` save is current state and execute normally. Otherwise an alarm is raised to inform that the execution flow have been diverted. This is explained in detail in section 6.3.

Toolchain

All metadata are now computed during the compilation, by adding a plugin to GCC and modifying the linker. The creation of metadata is done in two phase during the assembly and the linkage. Figure 6.3 summarizes the compilation flow.

Firstly a GCC plugin is inserted to the compilation workflow to insert assembly directive in the assembly code generated to add metadata. For the compiler a BB can contain a `call`, this is not compatible with our approach. During this phase BB containing call are split up to be compatible with

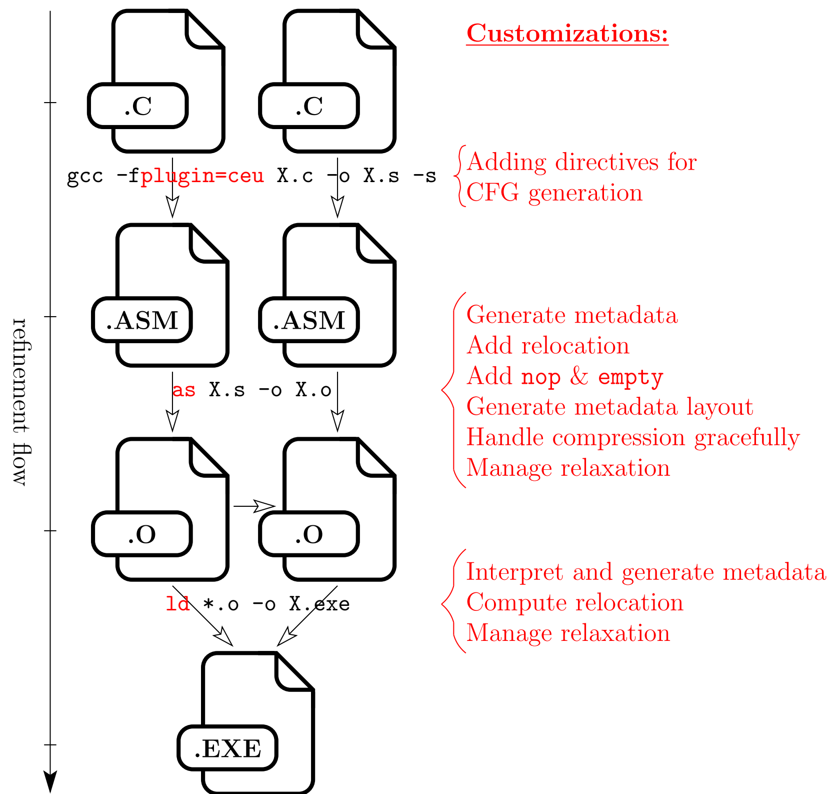


Figure 6.3: Compilation flow

our solution. When a BB is too small to contain all needed metadata a corresponding numbert of `nop` is added to extend it. This case happens generally when a BB is only one instruction or when the end of the BB is a indirect jump with a lot of destination possible. At the end of this first phase memory space have been allocated to store metadata for each BB but there values are not yet set.

The second phase is done with the linker, when all relocation are done all `VD` in metadata can be filled. The signature `Hash` is also computed at this stage, after the relocation stage because the linker can modify some instruction like short jump and long jump.

6.2 Speculative execution

Depending on the specific implementation of the CPU, there can be other situations that require adjustment of the solution. One such case is branch prediction and speculative execution, which leads to a mismatch between the *fetched* instructions and those that are effectively *executed*. Since the CFI-cache monitors the cache interface of the CPU, it needs to be aware of such features in order to correctly detect the destination of branches. This section describe how such feature is detected as an attack by a simple implementation and explain needed change on the CCFI-Checker to be able to protect processor using speculative execution. Section 6.4 explains how we have resolved this issue for the used RISC-V implementation.

Speculative execution is commonly used in modern processors, and even in some microcontrollers due to its benefits in term of performance improvement. From our CFI point of view speculative execution can be detected as CF violation since the processor actually begins to execute some instructions of the predicted jump destination. If the branch prediction appears to be incorrect, it will rollback all change induced by the speculative execution and jump to the right address. This behaviour is represented on figure 6.4, where the processor have predicted the that the branch will no be taken and execute speculatively the instruction at the address 133*c*. Prediction was wrong, and processor rollback and jump to address 1210. This impromptu jump is detected as violation of the CFG by the basic implementation presented in the previous section 6.1. More advanced processor, in order to improve per-

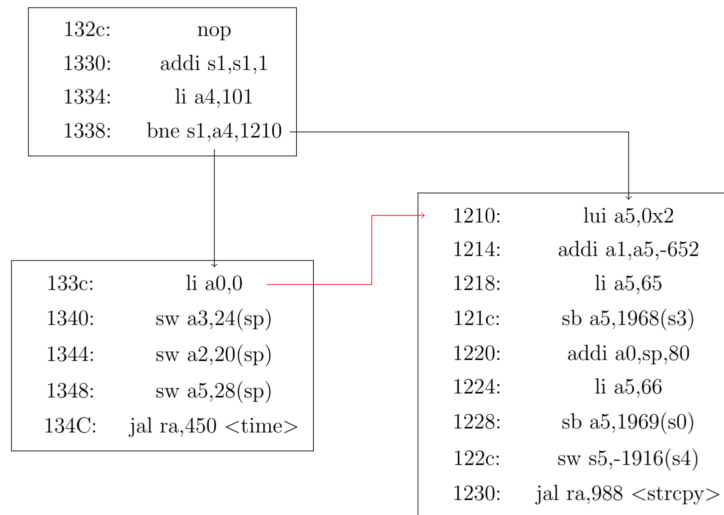


Figure 6.4: Exemple of miss branch prediction

formance, can also have prefetch technique that make it impossible to follow the execution flow from instruction fetched.

To address this behaviour we have to make distinction between instruction fetched and instruction executed. Each instruction fetched is stored in a circular buffer along with its metadata. This circular buffer is as deep as needed to reproduce the latency of the pipeline between fetch stage and the stage of trace interface. For each fetched instruction by the processor, one line is stored in the `Buf` module. Each line contain the instruction, its address and the metadata associated. So when the trace interface output an address, this address is send to `Buf` module to select the corresponding line. By construction it is never possible for the processor to output an PC from the trace interface without having its corresponding metadata present in `Buf`. Once the line selected, `Buf` the related information (PC, instruction and metadata) to the `CCFI-Checker`. Doing so `CCFI-Checker` is able to follow the execution of the processor step by step without error even with an processor using speculative execution.

This solution for handling speculative execution is easily scalable on different sizes of processor pipeline and prediction mechanism by adjusting the maximum size of the cyclic buffer. This approach also has the advantage of not limiting the number of valid destination we can store for one BB.

6.3 Interruptions

In the literature of CFI, interruption and exception are rarely discussed due to the fact they can happen at any time and break the CFG of the current running program.

When an interruption occurs the processor determines the memory address of the interruption handler and jumps to it. There two major ways for a processor to execute the handler. The first one is to have static hard-coded address in the processor, regardless of the interruption the processor will execute the code at this address. The distinction of the type of interruption and the call of the right function handler is left to the programmer. The second method is to have a dedicated memory zone for an array of code pointer. For each one of interruption the processor fetches the corresponding code pointer and executes the pointed handler.

In all cases this results to jump directly on the interruption subroutine at any time and from anywhere. From outside of the processor this behaviour is viewed as a violation of the CFG. To be agnostic of the type of interruption our solution consists in adding an Int flag in the metadata header of the first and last BB of the handler function to detect start and end of handler function.

This allows to detect on-the-fly triggered interruptions, regardless of the processor implementation. Upon interruption, CCFI-checker detects the discontinuity of the control flow but metadata will indicate this is at the same time an Start BB end Int meaning this function is call because of an interruption. Figure 6.5 is a partial view of the `CCFI-Checker` FSM. From any state if the next instruction is the Start of a BB and also the beginning of an interruption procedure so `CCFI-Checker` save its current context of execution in an internal memory. The shadow stack is used to save the current PC. Once it is done the `CCFI-Checker` jumps to another FSM dedicated to follows the execution of the interruption handler (right FSM of the figure 6.5). This FSM is the same that the normal FSM except for the `END BB` where the flag `ENDINT` is checked. If the `ENDINT` flag is present in a `ENDBB` then the previous saved context is restored. Once the context have been restored CCFI-checker can continue verification of the BB where it has been interrupted.

This mechanism of saving and restoring context allows to be interrupted

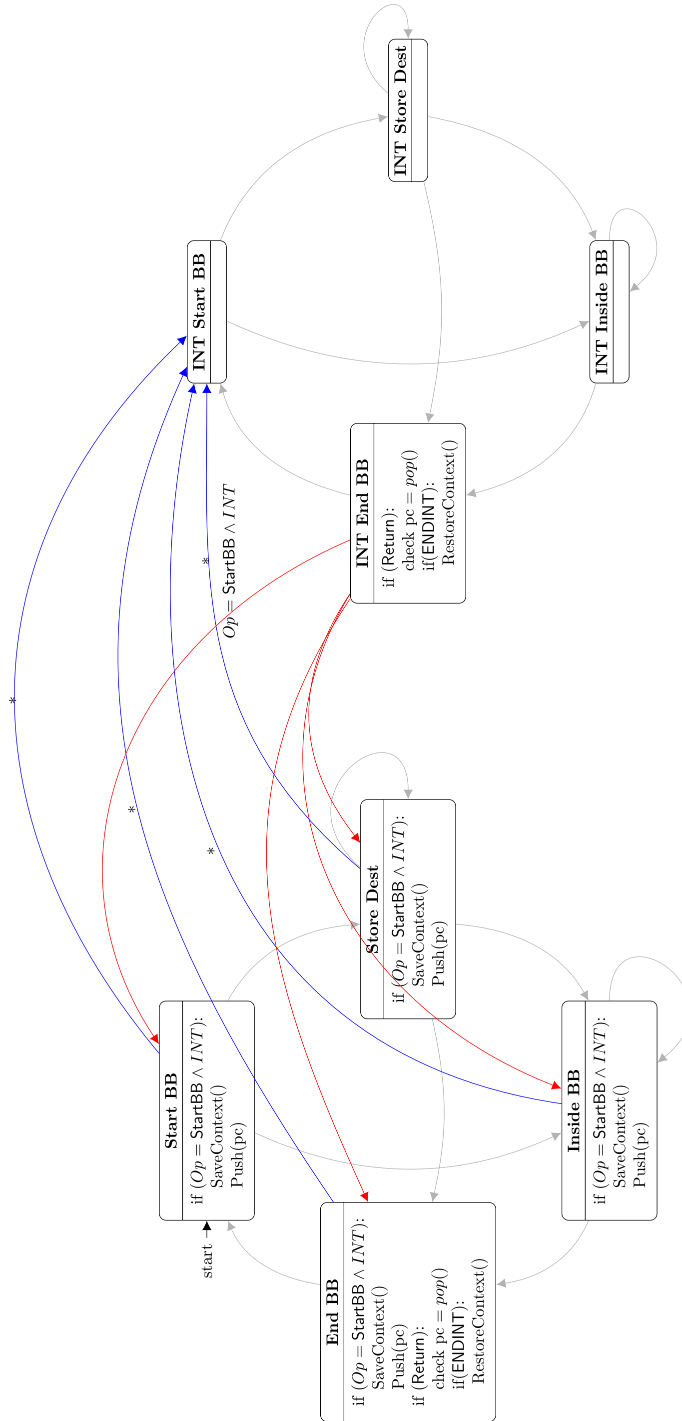


Figure 6.5: Interruption FSM

in the middle of a BB without triggering any false CFI alarms. In case of reentrancy a internal stack is used to save multiple contexts. Reentrance level is limited by the depth of this stack.

To overcome this limitation, next implementations this stack can trigger an alarm to flag when its full. This will allow the programmer to implement a routine to dump this stack in memory. This virtually allow to have unlimited interruption reentrancy. This technique will introduce a vulnerability in our design since data of the internal state of CCFI-checker will be exposed to the monitored program.

6.4 Implementation

In order to validate the CCFI architecture, we have implemented it on microcontroller platform based on industrial RISC-V processor [11]. The CPU core is a 5-stage pipeline processor with a two memory interfaces for accessing instructions and data (Harvard architecture). The platform uses a crossbar for memory access, with one ROM for code and metadata and one RAM for the execution.

As mentioned in Section 6.2, branch prediction can potentially pose a problem for the CCFI-checker, since the address seen on the memory interface (i.e. the program counter) may not coincide with the effective branch target address. The targeted processor have branch prediction and speculative execution capabilities. It also had a complex prefetch system which prevent us to directly use the PC from the instruction cache fetch to determine the execution flow of the program. We claim that the proposed architecture is suitable for complex prediction schemes, A detailed discussion of the required modifications is beyond the scope of this paper.

6.5 Performance

The interruption support has been tested in many benchmarks having predictable and random interruptions. Our implementation is able to save and restore context of execution without errors and significant impact on the performance level.

Table 6.1: Hardware cost

Component	**Cells**	**Cell Area**
CCFI-cache	2078	58728
CCFI-checker	8199	24812

Table 6.2: Hardware cost of caches

Component	**Cells**	**Cell Area**	**Net Area**	**Total**
Dcache ctrl	3808	63278	6194	69473
Icache ctrl	2114	58655	3511	62166

Table 6.1 and 6.2 report the hardware cost of CCFI-cache module, CCFI-checker and caches on Intel's Cyclone V FPGA.

Table 6.3 presents the execution time in number of cycles when the code is running on a more powerful processor with speculative execution. Nominal run is the software without any modification and without the presence of CCFI. NOP only corresponds to the software modified by the toolchain but without the CCFI module. This test gives us the impact on performance when adding NOP to enlarge BB. CFI/NOP corresponds to the run fully protected by CCFI module. These benchmarks show an impact of 21% on average. Most of the overhead is due to the NOP added in the code to ensure metadata alignment.

As we can see in table 6.3 nearly all runtime overhead came from the added `NOP`. In total `NOP` added represent 25% of more code. Empty metadata represented the memory space allocated for metadatas not used. As we can see around 31% of memory space of metadata does not store useful information.

6.6 Conclusion

In this chapter we present a more complex approach of the CCFI solution to handle advanced processor using speculative execution and branch prediction. We also present metadata and hardware modification to be able to detect and protect interruptions. This solution remains independent of the

Table 6.3: Benchmark on A* processor in number of cycle

Run	**Bubble Sort**	**Drystone**	**AES**
Nominal	1023533	420857	6342603
CFI/NOP	1289903	443337	8340834
NOP only	1289599	442644	8339606
Software overhead	25.99%	5.17%	31.48%
Hardware overhead	0.02%	0.15%	0.01%
Total overhead	26.02%	5.34%	31.50%

Table 6.4: Software modification

Run	**Bubble Sort**	**Drystone**	**AES**
number total of instructions	989	1490	2944
number total of BB	251	339	528
number total of nop added	216	232	456
number total of empty metadata	327	575	1127

processor although it needs a specific trace interface from the processor to be able to work. However, modifying the processor would be error-prone since revalidation is costly, and legacy processors cannot be modified anyway.

Performance of the solution remains the same as the previous CCFI architecture while increasing the size of the needed hardware. The bottleneck of our solution remain is that the binary code shall be instrumented with some extract NOPs to match the size of basic blocks in respectively the code and the metadata. As a perspective, the compiler shall be involved actively to help produce basic blocks which are not too small, thereby reducing the overhead caused by such stuffing.

Profiling attest that this solution is very competitive regarding the hardware overhead and the performance penalty, which are minimal and affordable in most cases which make our technology practical and deployable.

Chapter 7

Conclusion

Code and control flow integrity checking in parallel with the software execution is of utmost importance for security and safety applications.

The base idea of using cache technologies to retrieve metadata simultaneously with the code has been proven to work well, performance-wise, while not hindering security. Indeed, alignment of metadata with the code structure has proved to be a good idea both to simplify the computation of the location of metadata from code's address and to make the best use of the metadata cache.

We have proven this is possible develop the current finest-grain solution for CFI while maintaining good performance, with the help of dedicated hardware (CCFI-Cache). One major advantage of our solution is its reduced size in silicon. Namely, it is not dependent on the power of the processor. Still, some specialized IPs are required in addition to the design to work with complex processors, for example those using speculative execution.

The overhead cost of adding twice the size of the (cache) code can be seen controversial among industrialists and academics. However, it can be said that it is a reasonable price to pay for security and today's memory is relatively cheap. Besides, when code is compiled efficiently, it remains in the cache, therefore runtime speed is not compromised. In other use-cases, it can be said that it is cost-sensitive to double the size of memory especially when this memory is embedded within the SoC.

As we have seen during the development of the CCFI-Cache solution, one of the challenges is to be able to detect the beginning and the end of basic block online. Indeed during the execution, only instructions that change the control flow give hint of the CFG but this is not enough to deduce the full CFG. We have smartly circumvented this problem by adding these pieces of information directly in our metadata.

In a nutshell, we underline that the solution ensures CFI at the basic block level, which makes it efficient and capable of catching all attacks modifying the program execution flow. Solution's impact on performance are promising varying from 2% to 30% depending on the type of program and the architecture put in place, which is very reasonable and industrially viable.

Chapter 8

Perspectives

In this chapter we propose some research perspectives to improve CCFI solution. These tracks are base on the previous results of our research. First we summarize limitations of our approach and then we propose some idea to improve performances and flexibility of the solution.

8.1 Limitation of our approach

Compared to state-of-the-art solutions, we have shown that our solution has the largest coverage against SOTA solution. But we do not detect:

- **Change on non-control-data**: In this scenario, the attacker changes non-control-data using software bug or using physical attack targeting memory or the processor. This class of attack is undetected because the data value are altered while respecting the CFG.

- **Correlated alteration on code and metadata**: In this sophisticated attack, the attacker is able to modify the code and at the same time modify metadata accordingly to respect rules of *CCFI-checker*.

8.2 Interface with the processor

Our choice to implement a hardware CFI solution without modifying the processor pipeline was a challenge. The choice to connect between the processor and its caches have demonstrated some limitation, especially when processor use speculative execution or branch prediction.

There are two way to overcome difficulties linked to speculative execution. Both implies to extract information from the pipeline, without modification of the logic.

First is to have more information from the fetch stage like:

- Instruction Fetch
- Program Counter
- When it is speculative execution

This solution allows us to follow the CFG even if there is a prefetch unit. With this approach CCFI must keep heuristic branch prediction model. This approach may not work with processor having large number of stages in its pipeline. In fact CCFI will have to reproduce the delay between fetch and instruction commit, drastically increasing the silicon area needed.

The second solution will be to extract information of executed instruction from the write back stage. With this approach we are able to follow more precisely the CFG of the program since write back stage only commits instruction really executed. This introduces a delay of the size of pipeline in the verification process of the BB. In return we do not have to implement dependant heuristic to match the processor's branch prediction. Unfortunately it is common for processor implementation to drop fetch instruction after the decode stage. Meaning it is not possible anymore to retrieve information past this stage.

There is actually a draft proposal for a standardisation of a trace interface for RISCV (https://github.com/riscv/riscv-trace-spec). This proposition is born from the difficulty of understanding program behavior. In fact some programs are difficult to monitor due to processor behavior and event like

realtime events, exception, interruption or multiprocess programming. Usually software debugger is used to follow the execution of a process, this is based on using special instructions inserted on the code. This technique is considered as intrusive and can show some limitation. We can imagine in the future that interface will give enough information (instruction executed, program counter, conditional branch taken or not, ...) to be used by our solution to both simplify the complexity of CCFI and be adaptable to complex processor.

8.3 Metadata alignment

Our idea to align metadata with the code is a good idea to solve the trouble to find the corresponding metadata in memory. Experiment show that approach to work but at cost of doubling the memory space unused by the code. Opinions on this cost vary, some think it is a cheap cost to pay for this level of security, especially because memory is cheap and in general code size is small compare to data size. Other think this cost is too high when we need to double the size of code memory in SOC, which is more expensive than external memory.

In our experimentation half of the metadata are empty and therefore useless, so there is room for optimization. To resolve this problem we need to improve how we compute address of metadata from instruction's address. One idea would be to find a function f which given a address of the start of a BB $@BB$ is able to compute the corresponding metadata address $@META$. This would give the following function $f(@BB) = @META$. One of the challenges of this idea is that this function wouldn't be linear.

Another approach can be to store, with the address of valid destination, the address of the corresponding metadata if this destination is taken. This would enlarge the metadata but will get ride of the need of empty metadata. However we will have a problem when a BB do not have valid destination inside his metadata.

8.4 Cache

One other optimisation is to merge metadata cache with instruction cache. This will reduce the cost of a miss but imply major modifications of the platform architecture such as the bus and memory size. The advantage to merge the instruction cache and the metadata cache will be to reduce the latency of fetching metadata and to reduce the size of silicon by merging caches logic.

8.5 Basic Block Optimisation

One of the impact on performance is to add NOP instructions when a BB is to small to contain all needed metadata. One approach to reduce the number of inserted NOP will be to modify the compiler so that is does not generate small BB. It can be done by merging BB together event if it mean duplicating the code. It will be a tread off between the code size and the performance at the execution.

Acronyms

AMBA Advanced Microcontroller Bus Architecture.

BB Basic Block.

CFG Control Flow Graph.

CFI Control Flow Integrity.

CHERI Capability Hardware Enhanced RISC Instructions.

CI Code Integrity.

CRC Cyclic Redundancy Check.

ELF Executable and Linkable Format.

FIFO First In First Our.

FSM Finite State Machine.

ISR Instruction Set Randomization.

JOP Jump Oriented Programming.

MAC Message Authentication Code.

OOO Out-of-Order.

PC Program Pointer.

RISC Reduced Instruction Set Computer.

ROP Return Oriented Programming.

Chapter 9

Bibliography

[1] Introduction to Intel® Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2016.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[5] ADM. AMD64 Virtualization Codenamed "Pacifica" Technology. Secure Virtual Machine Architecture Reference Manual. http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[6] Alejandro Cabrera Aldaya, Alejandro Cabrera Sarmiento, and Santiago Sánchez-Solano. AES t-box tampering attack. *J. Cryptographic Engineering*, 6(1):31–48, 2016.

[7] Zeyad Alkhalifa, V. S. S. Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):627–641, 1999.

[8] AMD. AMD Memory Ecryption, White paper. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`.

[9] Anonymous. Once upon a free()... *Phrack 57*, page 9, 08 2001.

[10] ARM. ARM TrustZone. `https://www.arm.com/products/security-on-arm/trustzone`.

[11] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[12] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In *CRYPTO*, volume 5157 of *LNCS*, pages 221–240. Springer, 2008. Santa Barbara, CA, USA.

[13] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO*, volume 1294 of *LNCS*, pages 513–525. Springer, August 1997. Santa Barbara, California, USA. DOI: 10.1007/BFb0052259.

[14] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 30–40. ACM, 2011.

[15] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[16] Erik Bosman and Herbert Bos. Framing signals - A return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy, SP 2014,*

Berkeley, CA, USA, May 18-21, 2014, pages 243–258. IEEE Computer Society, 2014.

[17] Nicholas Carlini and David A. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, 2014.*, pages 385–399, 2014.

[18] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY, New Orleans*, pages 38–49, 2016.

[19] CODENOMICON. The heartbleed bug. `http://heartbleed.com/`.

[20] TIS Comitee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification v1.2*, 1995.

[21] Thomas Coudray, Arnaud Fontaine, and Pierre Chifflier. PICON: control flow integrity on LLVM IR. In *Symposium sur la sécurité des technologies de l'information et des communications, Rennes, France, June 3-5, 2015*, 2015.

[22] Jean-Luc Danger, Adrien Facon, Sylvain Guilley, Karine Heydemann, Ulrich Kühne, Abdelmalek Si-Merabet, and Michaël Timbert. Ccficache: A transparent and flexible hardware protection for code and control-flow integrity. In Martin Novotný, Nikos Konofaos, and Amund Skavhaug, editors, *21st Euromicro Conference on Digital System Design, DSD 2018, Prague, Czech Republic, August 29-31, 2018*, pages 529–536. IEEE Computer Society, 2018.

[23] Jean-Luc Danger, Sylvain Guilley, Thibault Porteboeuf, Florian Praden, and Michaël Timbert. HCODE: hardware-enhanced real-time CFI. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, New Orleans*, 2014.

[24] Jean-Luc Danger, Sylvain Guilley, Thibault Porteboeuf, Florian Praden, and Michaël Timbert. Hardware-Enforced Protection Against Buffer Overflow Using Masked Program Counter. In Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater, editors, *The New Codebreakers*

- *Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, volume 9100 of *Lecture Notes in Computer Science*, pages 439–454. Springer, 2016.

[25] Jean-Luc Danger, Sylvain Guilley, and Florian Praden. Hardware-enforced protection against software reverse-engineering based on an instruction set encoding. In Suresh Jagannathan and Peter Sewell, editors, *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2014, PPREW 2014, January 25, 2014, San Diego, CA*, pages 5:1–5:11. ACM, 2014.

[26] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 133:1–133:6. ACM, 2014.

[27] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen De Bosschere, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. SOFIA: software and control flow integrity architecture. In Luca Fanucci and Jürgen Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1172–1177. IEEE, 2016.

[28] Solar Designer. lpr LIBC RETURN exploit. `http://insecure.org/sploits/linux.libc.return.lpr.sploit.html`.

[29] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 103–114. ACM, 2008.

[30] Digilent. Nexys4 DDR FPGA board reference manual, april 2016.

[31] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In Darrell M. Kienzle, Mary Ellen Zurbo, Steven J. Greenwald, and Cristina Serbau, editors, *Proceedings*

of the 1999 Workshop on New Security Paradigms, Caledon Hills, ON, Canada, September 22-24, 1999, pages 87–95. ACM, 1999.

[32] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 246–255. IEEE Computer Society, 2000.

[33] GlobalPlatform. GlobalPlatform made simple guide: Trusted Execution Environment (TEE) Guide. `http://www.globalplatform.org/mediaguidetee.asp`.

[34] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2003), 3-5 November 2003, Boston, MA, USA, Proceedings*, pages 581–588. IEEE Computer Society, 2003.

[35] IlyaEnkovich. Intel® Memory Protection Extensions (Intel® MPX) support in the GCC compiler. `https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler`, 2016.

[36] Intel. Control-flow Enforcement Technology Preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

[37] Intel. Intel SGX. `https://software.intel.com/en-us/sgx`.

[38] Intel. Control-flow Enforcement Technology Preview, Revision 2.0, June 2017.

[39] JonathanSalwan. Ropgadget. `https://github.com/JonathanSalwan/ROPgadget/tree/master`.

[40] Marc Joye and Michael Tunstall. *Fault Analysis in Cryptography*. Springer LNCS, March 2011. `http://joye.site88.net/FAbook.html`. DOI: 10.1007/978-3-642-29656-7 ; ISBN 978-3-642-29655-0.

[41] jp. Advanced doug lea's malloc exploits. *Phrack 61*, page 6, 08 2003.

[42] Michel "MaXX" Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack 57*, page 8, 08 2001.

[43] D. Karaklajić, J. M. Schmidt, and I. Verbauwhede. Hardware Designer's Guide to Fault Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, December 2013.

[44] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, 2014.

[45] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and André DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In Sadeghi et al. [61], pages 721–732.

[46] Laginimaineb. Bits, please!: Full trustzone exploit for msm8974. `http://bits-please.blogspot.fr/2015/08/full-trustzone-exploit-for-msm8974.html`.

[47] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card C codes. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 200–218. Springer, 2014.

[48] longld. Ropme. `http://www.vnsecurity.net/research/2010/08/13/ropeme-rop-exploit-made-easy.html`, 8 2010.

[49] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In David R. Kaeli and Tipp Moseley, editors, *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, page 175. ACM, 2014.

[50] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In Jan Vitek and Doug Lea, editors, *Proceedings of the 9th International Symposium*

on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010, pages 31–40. ACM, 2010.

[51] Santosh Ganapati Nagarakatte. *Practical low-overhead enforcement of memory safety for C programs*. PhD thesis, Faculties of the University of Pennsylvania, 2012.

[52] NIST/ITL/CSD. Advanced Encryption Standard (AES). FIPS PUB 197, Nov 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[53] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[54] Aleph One. Smashing the stack for fun and profit. *Phrack 49*, page 14, 11 1996.

[55] Oracle. Oracle's SPARC T7 and SPARC M7 Server Architecture. `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/sparc-t7-m7-server-architecture-2702877.pdf`.

[56] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. ASIST: architectural support for instruction set randomization. In Sadeghi et al. [61], pages 981–992.

[57] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. ASIST: architectural support for instruction set randomization. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 981–992. ACM, 2013.

[58] Mathias Payer and Thomas R. Gross. String oriented programming: when ASLR is not enough. In Jeffrey Todd McDonald and Mila Dalla Preda, editors, *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*, pages 2:1–2:9. ACM, 2013.

[59] Phantasmal Phantasmagoria. The malloc maleficarum, 10 2005.

[60] Rajp-Oracle. Silicon Secured Memory - It's Better Than You Might Think. https://blogs.oracle.com/raj/entry/silicon_secured_memory_in_action.

[61] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013.

[62] Jonathan Salwan. Shellcodes database. http://shell-storm.org/shellcode/.

[63] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 745–762. IEEE Computer Society, 2015.

[64] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561. ACM, 2007.

[65] Di Shen. Attacking your "Trusted Core" Exploiting TrustZone on Android. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf, 2015.

[66] sslab.gtisc.gatech.edu. Plateforme libre OpenSGX :. https://github.com/sslab-gatech/opensgx.

[67] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In Shubu Mukherjee and Kathryn S. McKinley, editors, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 85–96. ACM, 2004.

[68] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.

[69] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.

[70] Michaël Timbert, Jean-Luc Danger, Adrien Facon, Sylvain Guilley, Karine Heydemann, Ulrich Kühne, Abdelmalek Si Merabet, and Baptiste Pecatte. Processor Anchor to Increase the Robustness against Fault Injection and Cyber Attacks. In *Constructive Side-Channel Analysis and Secure Design - 11th International Workshop, COSADE 2020, Lugano, Switzerland, October 5, 2020, Proceedings*, LNCS. Springer, 2020.

[71] Victor van der Veen, Dennis Andriesse, Enes Göktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver*, pages 927–940, 2015.

[72] Robert N.M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Philip Paeps, Michael Roe, and Hassen Saidi. CHERI: a research platform deconflating hardware virtualization and protection. Unpublished workshop paper for Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE), 2012.

[73] Mario Werner, Erich Wenger, and Stefan Mangard. Protecting the control flow of embedded processors against fault attacks. In *Smart Card Research and Advanced Applications (CARDIS), Bochum. Revised Selected Papers*, pages 161–176, 2015.

[74] Clifford Wolf. PicoRV32 - A size-optimized RISC-V CPU.

[75] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 457–468. IEEE Computer Society, 2014.

ECOLE DOCTORALE

Titre : Protections des processeurs contre les cyber-attaques par vérification de l'intégrité du flot d'exécution

Mots clés : Processeur, Intégrité de flot de contrôle, Cyber Sécurité, Cyber attaques

Résumé : Les cyber-attaques reposent sur l'intrusion des systèmes numériques en exploitant des vulnérabilités pour prendre le contrôle du système. De nombreuses protections existent contre les cyber-attaques. Parmi elles, citons les techniques d'obfuscation de code, de vérifications d'intégrité de la mémoire, la personnalisation du jeu d'instruction, la distribution aléatoire de l'espace d'adressage (ASLR), les anticipations par les canaris ou bac à sable, l'isolation des processus (machines virtuelles), la gestion de droits d'accès. Au niveau matériel, les processeurs modernes procurent des techniques de sécurisation par isolation de zones (anneaux de protection, MMU, NX bit, Trustzone). Le Contrôle de l'Intégrité du flux d'exécution (Control Flow Integrity, CFI) est une nouvelle technique proposée par Abadi et al. pour empêcher la corruption d'un programme. Cette technique a donné lieu à beaucoup d'implémentations mais aucune n'est à la fois complète, rapide et facilement incorporable aux processeurs existants. Cette thèse est inspirée des travaux de HCODE qui implémente l'intégrité du code par calcule de signature pour chaque bloc de base de code exécuté. HCODE est un module matériel conçu pour être connecté en lecture seule sur l'interface entre le processeur et le cache d'instruction. Dans cette thèse nous présentons une amélioration de HCODE nommée CCFI qui fournit à la fois la protection de d'intégrité de code et l'intégrité du flux d'exécution. Nous proposons une architecture capable de protéger les sauts directs et indirects aussi bien que les interruptions. La solution proposée repose à la fois sur des modules matériels et sur des modifications du code pour assurer rapidité et flexibilité de la solution. Pour garantir une protection CFI complète, des métadonnées sont ajoutées au code. Ces métadonnées décrivent le graphe de flot de contrôle (Control Flow Graph, CFG) du programme. Celles-ci sont calculées statiquement pendant la phase de compilation et sont utilisées par le module matériel CCFI en conjonction avec le code exécuté pour garantir que le CFG est respecté. Nous démontrons que notre solution est capable de fournir une intégrité du flux d'exécution Complète en étant à la fois rapide et facilement adaptable aux processeurs existants. Nous l'illustrons sur deux processeurs RISC-V.

Title : Protections of Processors against Cyber Attacks by Control Flow Checking

Keywords : Processor, Control Flow Integrity, Cyber Security, Cyber attacks

Abstract : Cyber attacks are based on intrusions into digital systems by exploiting bugs to take control over the system. Many protections have been developed to thwart cyber attack, among them we can quote code obfuscation, memory integrity check, instruction set randomization, address space layout randomization (ASLR), canary, sand boxing, process isolation, virtualization and access right restriction. Modern processors provide security by zone isolation systems (Protection ring, MMU, NX bit, TrustZone), Control Flow Integrity (CFI) is a new technique proposed by Abadi et al. to mitigate program corruption. This technique gave rise to many implementations but none are complete, fast and easily incorporable to existing processor. This thesis is inspired from previous work on HCODE which implements code integrity by computing signature for each executed basic block. HCODE is an hardware block designed to be plugged in read only on the interface between the processor and the instruction cache. In this thesis we present CCFI solution, improvement of HCODE, which is now able to provide Code Integrity and Control Flow Integrity. We propose CCFI architecture able to protect direct and indirect jumps as well as interruptions. The proposed solution is based on both hardware modules and software modifications to ensure speed and flexibility of the solution. To ensure a full CFI protection metadata are embedded with the code. These metadata describes the Control Flow Graph (CFG) of the program. These are computed statically during compilation and are used by the hardware CCFI module in conjunction with executed code to ensure that the CFG is respected. We demonstrate that the our solution is able to provide a full CFI solution while being fast and easily adaptable to different processors. We illustrate it on two RISC-V Processor.

Institut Polytechnique de Paris
91120 Palaiseau, France