Sécurisation de l'exécution des applications contre les attaques par injection de fautes par une contre-mesure intégrée au processeur

Thomas Chamelot

Sorbonne Université
CEA

École doctorale **Informatique, Télécommunications et Électronique**

Unité de recherche **CEA List**

Thèse présentée par **Thomas Chamelot**

En vue de l'obtention du grade de docteur de Sorbonne Université

Spécialité **Informatique**

Sécurisation de l'exécution des applications contre les attaques par injection de fautes par une contre-mesure intégrée au processeur

Sécurisation de l'exécution des applications contre les attaques par injection de fautes par une contre-mesure intégrée au processeur

Résumé

Les systèmes embarqués numériques sont omniprésents dans notre environnement quotidien. Ces systèmes embarqués, par leur caractère nomade, sont particulièrement sensibles aux attaques dites par injection de fautes. Par exemple, un attaquant peut injecter une perturbation physique dans un circuit électronique pour compromettre les fonctionnalités de sécurité du système. Originellement utilisées pour compromettre des systèmes cryptographiques, ces attaques permettent aujourd'hui de cibler n'importe quel type de système. Ces attaques permettent notamment de compromettre l'exécution d'un programme.

Dans ce manuscrit, nous introduisons une nouvelle propriété de sécurité pour protéger l'exécution des instructions dans la microarchitecture : *l'intégrité d'exécution*. À partir de cette propriété, nous décrivons le concept de SCI-FI, une contre-mesure qui assure la protection de l'intégralité du chemin d'instructions en assurant l'intégrité du code, du flot de contrôle et d'exécution. Pour cela, nous construisons un vecteur de bits que nous appelons *pipeline state* à partir de signaux de contrôle dans la microarchitecture. À partir du pipeline state, deux modules s'articulent pour assurer les propriétés de sécurité. Le premier module calcule une signature à partir du pipeline state assurant ainsi l'intégrité du code, du flot de contrôle et une partie de l'intégrité d'exécution. Le second module complète l'intégrité d'exécution dans la microarchitecture en utilisant un mécanisme de redondance. Nous proposons également le support et la sécurisation des branchements indirects et des interruptions, nécessaires pour la conception de systèmes embarqués.

Nous réalisons deux implémentations de SCI-FI, l'une construite sur une primitive cryptographique assurant un niveau de sécurité maximal et l'autre plus légère construite sur une fonction CRC privilégiant les performances. Pour cela, nous intégrons SCI-FI dans un processeur RISC-V 32 bits et modifions la chaîne de compilation LLVM. Nous réalisons une analyse de sécurité des différents éléments qui composent SCI-FI dans chaque implémentation. Nous montrons ainsi que SCI-FI, même avec l'implémentation privilégiant les performances, est robuste face à un attaquant disposant de moyens d'injection de fautes à l'état de l'art. Enfin, nous évaluons les performances de nos implémentations par une synthèse dans un flot de conception ASIC et par l'exécution en simulation de la suite de test Embench-IOT. Nous montrons ainsi que SCI-FI a des performances équivalentes aux contre-mesures de l'état de l'art tout en assurant une propriété de sécurité supplémentaire, l'intégrité d'exécution.

Mots clés : injection de fautes, contre-mesure, microarchitecture

Abstract

Embedded systems are ubiquitous in our everyday life. Those embedded systems, by their nomadic nature, are particularly sensitive to the so-called fault injection attacks. For example, an attacker might inject a physical perturbation in an integrated circuit to compromise the security features of the system. Originally used to compromise cryptographic systems, those attacks can now target any kind of system. Notably, those attacks enable to compromise the execution of a program.
In this manuscript, we introduce a new security property to protect the execution of instructions in the microarchitecture: *execution integrity*. From this property, we describe the concept of SCI-FI, a counter-measure that ensures the protection of the whole instruction path thanks to code, control-flow and execution integrity properties. We build SCI-FI around a bit vector that we call *pipeline state* and that is composed of microarchitecture control signals. Two modules interact around the pipeline state to ensure the security properties. The first module computes a signature from the pipeline state to ensure code and control-flow integrity and partially execution integrity. The second module completes the execution integrity support in the microarchitecture thanks to a redundancy mechanism. We also propose a solution for indirect branches and interrupts that are required to design embedded systems. We implement two versions of SCI-FI, one built around a cryptographic primitive which provides the best security level and another lighter one built around a CRC to maximize the performances. We integrate SCI-FI into a 32 bits RISC-V processor, and we modify the LLVM compiler. We analyze the security provided by our two implementations and we show that SCI-FI, even with the lightweight implementation, is robust against state-of-the-art attacker. Finally, we evaluate the performances of our implementations through an ASIC synthesis and through the execution of the benchmark suite Embench-IoT. We show that SCI-FI has comparable performances to state-of-the-art counter-measures while ensuring a new security property: execution integrity.

Keywords: fault injection, counter-measure, microarchitecture

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

Sommaire du présent chapitre

1.1 Introduction

Depuis maintenant plusieurs années, les systèmes numériques sont omniprésents dans notre environnement quotidien et leurs nombres augmentent fortement avec la transformation numérique. Au quotidien, un individu utilise de nombreux systèmes numériques pour accéder à différents services. Certains systèmes manipulent des données sensibles comme les cartes bancaires, les cartes vitales ou les passeports. D'autres traitent des données liées à la vie privée comme les smartphones et autres objets connectés. Enfin, de nombreux systèmes embarqués jouent un rôle critique dans notre quotidien comme l'électronique dans les véhicules.

Un attaquant peut essayer de voler ou de bloquer l'accès à des données avec pour objectif de faire du chantage. C'est, par exemple, le cas des rançongiciels qui, pour le moment, ciblent plutôt les systèmes d'information classiques. Un autre objectif peut être de voler l'identité numérique d'une personne pour effectuer des actions en son nom, telles que des transactions bancaires. Enfin, les données privées ont aujourd'hui une forte valeur commerciale et un attaquant peut les revendre afin de s'enrichir. En ce qui concerne les systèmes critiques, une attaque peut mettre la vie des utilisateurs en danger.

La sécurité des systèmes embarqués est un enjeu majeur. C'est pourquoi les institutions gouvernementales imposent à ces systèmes de répondre à différentes normes avant d'autoriser leur commercialisation, par exemple les critères communs [Criteria 2017]. Ces systèmes embarqués, par leur caractère nomade, sont particulièrement sensibles aux attaques dites matérielles. Ces attaques se déclinent en deux catégories :

— Les attaques par canal auxiliaire [Mangard et al. 2007]. Ce type d'attaque exploite le lien entre la consommation électrique d'un circuit et les données qu'il manipule pour retrouver des informations secrètes.
— Les attaques par injection de fautes [Yuce et al. 2018]. Ce type d'attaque exploite une perturbation physique injectée arbitrairement par l'attaquant dans un circuit. Cette perturbation entraine un mauvais calcul dans le circuit, ce qui permet à l'attaquant d'exploiter le résultat fauté par cryptanalyse, de contourner une authentification ou d'exécuter un programme arbitraire sur le circuit.

Ces dernières années, les attaques matérielles ont fait l'objet de nombreuses études et les capacités des attaquants ont fortement augmentées. Pour répondre à cette évolution des techniques d'attaques plusieurs contre-mesures ont été proposées. Premièrement des contre-mesures logicielles qui ont pour avantage de s'adapter aux systèmes numériques existant sans changer les composants matériels. Ces contre-mesures peuvent donc être déployées lors de la conception, mais aussi pendant le cycle de vie du produit si des mises à jour sont possibles. Cependant, les contre-mesures logicielles ne permettent pas de supprimer toutes les vulnérabilités. Avec l'objectif de produire de nouveaux systèmes dits sécurisés par conception, de plus en plus de travaux proposent d'intégrer les contre-mesures directement dans les circuits électroniques. Cette solution permet de considérer un plus grand nombre de vulnérabilités lors de la conception et donc d'assurer un meilleur niveau de sécurité. De plus, l'intégration des contre-mesures lors de la conception d'un système numériques permet souvent une meilleure intégration de la contre-mesure et donc un impact moindre sur les performances.

1.2 Problématique

Les attaques par injection de fautes sont devenues un sujet d'étude à part entière depuis les premiers travaux de Boneh et al. qui ont présenté l'attaque dite *Bellcore* en 1997 [Boneh et al. 1997]. Les attaques par injection de fautes ont d'abord été étudiées pour compromettre des implémentations de primitives cryptographiques réputées comme sûres mathématiquement. Avec les progrès faits dans le domaine des attaques par injection de faute, il est aujourd'hui possible d'attaquer une large variété d'applications au-delà des implémentations de primitives cryptographiques. C'est pourquoi les attaques par injection de fautes sont prises en compte lors des processus de certification, par exemple pour les niveaux EAL5 et supérieurs des Critères Communs [Criteria 2017].

Pour atteindre ces niveaux de certification, les systèmes numériques concernés s'appuient sur des contre-mesures à différents niveaux. Au niveau physique, des capteurs sont mis en place pour détecter les injections de fautes, par exemple des capteurs de température. Au niveau logique des contre-mesures assurent le bon comportement du système de son démarrage à l'exécution de ses fonctionnalités. Il est en particulier nécessaire de garantir que tout logiciel exécuté n'a pas été modifié de la mémoire le stockant au processeur l'exécutant. Il est aussi nécessaire de vérifier que l'exécution du logiciel ne dévie pas, c'est-à-dire que les instructions et l'ordre dans lequel elles sont exécutées sont conformes au programme d'origine. Enfin, il est nécessaire que l'exécution des instructions par le processeur soit correcte.

S'il existe une large panoplie de contre-mesures, elles sont souvent utilisées conjointement, mais leur combinaison est préférable. De plus, la bonne exécution des instructions par le por-

cesseur est souvent peu couverte. Pourtant, il a été montré par Laurent et al. que les attaques par injection de fautes peuvent provoquer des effets complexes dans la microarchitecture et que ces effets sont difficiles à capturer par les contre-mesures existantes [Laurent et al. 2019].

Ces travaux de thèse partent donc du constat qu'il est nécessaire de combiner les protections et d'y inclure la protection de la microarchitecture pour garantir la bonne exécution d'un programme. Il semble surtout judicieux et pertinent de concevoir de nouvelles contre-mesures qui regroupent plusieurs fonctionnalités pour couvrir complètement l'exécution d'un programme. Cela permettrait de diminuer la surface d'attaque grâce à la combinaison de contre-mesures tout en diminuant le coût dû à leur superposition.

1.3 Contributions

Dans ce manuscrit, nous introduisons une nouvelle propriété de sécurité pour protéger l'exécution des instructions dans la microarchitecture : nous l'appelons *l'intégrité d'exécution*. À partir de cette propriété, nous décrivons le concept de SCI-FI, une contre-mesure qui assure la protection de l'intégralité du chemin d'instruction en assurant l'intégrité du code, du flot de contrôle et d'exécution. Pour cela, nous construisons un vecteur de bits que nous appelons *pipeline state* à partir de signaux de contrôle dans la microarchitecture. À partir du pipeline state, deux modules s'articulent pour assurer les propriétés de sécurité. Le premier module calcule une signature à partir du pipeline state selon la technique *General Path Signature Analysis* (GPSA) assurant ainsi l'intégrité du code, du flot de contrôle et une partie de l'intégrité d'exécution. Le second module complète l'intégrité d'exécution dans la microarchitecture en utilisant un mécanisme de redondance. Nous proposons également le support et la sécurisation des branchements indirects et des interruptions, nécessaires pour la conception de systèmes embarqués.

Nous réalisons deux implémentations de SCI-FI, l'une construite sur une primitive cryptographique assurant un niveau de sécurité maximal et l'autre plus légère construite sur une fonction CRC privilégiant les performances. Pour cela, nous intégrons SCI-FI dans un processeur RISC-V 32 bits et modifions la chaîne de compilation LLVM. Nous réalisons une analyse de sécurité des différents éléments qui composent SCI-FI dans chaque implémentation. Nous montrons ainsi que SCI-FI, même avec l'implémentation privilégiant les performances, est robuste face à un attaquant disposant de moyens d'injection de fautes à l'état de l'art. Enfin, nous évaluons les performances de nos implémentations par une synthèse dans un flot de conception ASIC et par l'exécution en simulation de la suite de test Embench-IoT. Nous montrons ainsi que SCI-FI a des performances équivalentes aux contre-mesures de l'état de l'art tout en assurant une propriété de sécurité supplémentaire, l'intégrité d'exécution.

1.4 Plan du manuscrit

La suite de ce manuscrit est composée de quatre chapitres.

Le chapitre 2 présente un état de l'art sur les attaques par injection de fautes. Il commence par décrire les propriétés de sécurité associées à ces attaques. Ensuite, le processus d'attaques par injection de fautes est décrit. Enfin, il présente un état de l'art des contre-mesures logicielles

et mixtes matérielles logicielles contre les attaques par injection de fautes. La conclusion de ce chapitre pose la problématique à laquelle ces travaux de thèse répondent.

Le chapitre 3 présente notre proposition de contre-mesure pour protéger complétement le chemin d'instructions des attaques par injection de fautes. Pour cela, nous décrivons le principe de fonctionnement des deux modules matériels qui constituent cette contre-mesure. Nous présentons également les modifications matérielles et logicielles nécessaires pour supporter les fonctionnalités de branchement indirect et d'interruption.

Le chapitre 4 présente deux implémentations de notre contre-mesure. Il décrit les modifications apportées au processeur RISC-V 32 bits CV32E40P et à la chaine de compilation LLVM. Une analyse de sécurité évalue la robustesse de ces implémentations face aux attaques par injection de fautes. Nous évaluons également les performances matérielles et logicielles de ces implémentations.

Enfin, le chapitre 5 fait le bilan de nos travaux et nos contributions. Il propose également des axes d'amélioration de nos travaux et des idées à explorer pour compléter la protection d'un système contre les attaques par injection de fautes.

Chapitre 2

Attaques par injection de fautes

Sommaire du présent chapitre

2.1 Introduction

Cette thèse s'intéresse à la conception et à la mise en œuvre d'une contre-mesure contre les attaques par injection de fautes. Dans ce chapitre, nous présentons d'abord le fonctionnement des attaques par injection de fautes du mécanisme d'injection à l'exploitation de la faute. Ensuite, nous rapportons différents types de protection proposés dans l'état de l'art contre ces attaques.

2.2 Propriétés de sécurité

Dans le domaine de la sécurité des systèmes d'informations on considère quatre grandes propriétés de sécurité :

Confidentialité : l'information est uniquement accessible aux individus autorisés, par exemple pour protéger la vie privée ou une propriété intellectuelle ;

Intégrité : l'information n'a pas été modifiée (exactitude et complétude de l'information) par une action involontaire ou illégitime, par exemple lors de la diffusion de données sur un canal non sécurisé ;

Authenticité : l'information a été émise par une entité autorisée, et n'a pas été modifiée par une action involontaire ou illégitime, par exemple lors de la communication entre entités sur un canal non sécurisé ;

Disponibilité : l'information ou le service est et reste accessible aux utilisateurs dans le temps, par exemple pour l'accès à un service sur internet.

Pour un attaquant, l'objectif sera de compromettre ces propriétés de sécurité tandis qu'une protection cherchera à assurer ces propriétés. Suivant le besoin applicatif, il peut être nécessaire d'assurer une ou plusieurs de ces propriétés. Il est important de noter que, de par sa définition, l'authenticité implique l'intégrité.

Les attaques par injection de fautes sont souvent plus compliquées à mettre en place que les attaques informatiques classiques. Par exemple, il est plus simple d'attaquer la propriété de disponibilité d'un service en coupant l'alimentation du système qu'en réalisant une injection de fautes lorsque le système est accessible physiquement. C'est pourquoi un attaquant utilise l'injection de fautes en dernier recourt, lorsque les attaques informatiques classiques ne permettent pas de compromettre les propriétés de sécurité souhaitées.

2.2.1 Sécurité des données

L'objectif des systèmes numériques est de traiter automatiquement des données à la place des humains. Un attaquant cherchera toujours soit à récupérer une donnée secrète, soit à modifier le traitement d'une donnée pour accéder à un service dont l'accès est restreint. Il est donc important d'assurer la protection de ces données pour se protéger contre un attaquant.

Pour empêcher un attaquant d'accéder à des données secrètes, il est possible d'utiliser la propriété de confidentialité des données. Par exemple, la technologie Intel SGX [Costan et al. 2016] assure la confidentialité des données dans des systèmes multi-utilisateurs en chiffrant le contenu de la mémoire.

La protection contre la modification des données lors du stockage et du traitement est un problème complexe lié à l'intégrité des données. Dans les systèmes multi-utilisateurs, la virtualisation de la mémoire et l'utilisation de *memory management unit* (MMU) permet d'isoler les sections mémoires de chaque processus. Ainsi, un processus malveillant ne peut pas modifier le contenu de la mémoire d'un processus cible qui s'exécute sur le même système.

2.2.2 Sécurité du code

Pour traiter des données, une catégorie des systèmes numériques utilisent des processeurs. Ces circuits exécutent une suite d'instructions qui représentent le programme en charge du traitement des données.

Pour protéger le traitement des données, il est nécessaire de protéger le code via la propriété d'intégrité du code. Cela signifie qu'une protection s'assure que le code n'a pas été modifié avant

son exécution. Par exemple, de nombreux systèmes informatiques utilisent la fonctionnalité $W \oplus X$ pour empêcher les sections mémoires exécutables d'être modifiées par une écriture dans ces sections.

Il est également possible de vérifier que le code a bien été émis par une entité autorisée. Cette propriété est l'authenticité du code. On retrouve typiquement ce type de protection dans les mécanismes de démarrage sécurisé, pour lesquelles l'authenticité du code est vérifiée avant son chargement en mémoire pour exécution [Wang et al. 2022].

Enfin, le code qui représente un programme contient parfois des informations sensibles. Un attaquant peut utiliser de la rétro ingénierie pour voler un algorithme privé présent dans un programme ou construire une attaque. Il est donc parfois utile de limiter l'accès ou la compréhensibilité du code aux entités autorisées uniquement. Pour cela, il existe des techniques de dissimulation telles que l'obfuscation [Collberg et al. 2009] ou le code polymorphe [Sharma et al. 2014]. Ces techniques utilisent des procédés de réécriture de code pour complexifier sa compréhension par un humain. Il existe aussi des techniques de dissimulation à l'aide d'une fonction mathématique non cryptographique. Cela permet de stocker le code sous une forme dissimulée et de retrouver la valeur des instructions avant l'exécution. Ces techniques de dissimulation ne sont pas toujours robustes face à un attaquant. Les techniques de confidentialité du code permettent d'atteindre un meilleur niveau de sécurité basé sur des fonctions cryptographiques. Le code est conservé chiffré en mémoire et n'est déchiffré que lors de son exécution. Certains systèmes de démarrage sécurisé assurent ainsi la propriété de confidentialité du code [Wang et al. 2022].

2.2.3 Sécurité du flot de contrôle

Pour exécuter un programme, un processeur charge puis exécute les instructions de ce programme séquentiellement. Pour construire des structures de contrôle telles que les branchements conditionnels ou les boucles, certaines instructions permettent d'exécuter ensuite une autre instruction que celle qui suit séquentiellement. Le flot de contrôle d'un programme est l'enchainement des instructions qui composent ce programme. Le flot de contrôle est généralement représenté par un graphe orienté où les nœuds sont des séquences maximales d'instructions avec une seule instruction d'entrée et une seule instruction de sortie, appelées blocs de base. Les arcs sont les transitions entre les blocs de base. Le graphe de flot de contrôle (CFG) représente l'ensemble des chemins d'exécution possibles, pour un programme.

Plusieurs attaques exploitent une déviation du flot de contrôle pour exécuter arbitrairement du code. On peut par exemple citer le *Return Oriented Programming* (ROP) [Shacham 2007] et le *Jump Oriented Programming*(JOP) [Bletsch et al. 2011]. Pour se protéger contre ce type d'attaque, il est nécessaire de s'assurer que les chemins d'exécution pris par le programme sont bien définis dans le CFG. Abadi et al. proposent la propriété d'intégrité du flot de contrôle pour se protéger contre ce type d'attaque [Abadi et al. 2009]. On décompose généralement l'intégrité du flot de contrôle en plusieurs catégories [Burow et al. 2017] :

— intégrité des séquences d'instructions dans les blocs de base ;
— intégrité des branchements directs (destinations connues avant l'exécution) ;
— intégrité des branchements indirects (destinations inconnues avant l'exécution) ;
— intégrité des appels et retours de fonction.

Dans les systèmes d'information non embarqués, ce sont les retours de fonction et les branchements indirects qui sont à l'origine des vulnérabilités. Le caractère dynamique de ces transferts de flot de contrôle rend l'identification des adresses cibles complexes. Une technique courante pour identifier les cibles des branchements indirects est de regrouper les blocs de base par propriétés communes pour former des classes d'équivalence [Burow et al. 2017]. Chaque branchement indirect est ensuite associé à une unique classe d'équivalence et donc à un nombre fini de successeurs. Généralement, pour les appels de fonction indirects, les classes d'équivalence sont construites à partir des propriétés de fonctions (ensemble de blocs de base). Par exemple, il est possible d'utiliser une unique classe d'équivalence qui regroupe tous les premiers blocs de base des fonctions d'un programme. Tous les appels de fonction indirects sont associés à cette classe d'équivalence, ce qui signifie que toutes les fonctions peuvent être appelées depuis un branchement indirect. Un autre exemple est d'utiliser les prototypes de fonction pour regrouper les premiers blocs de base des fonctions d'un programme. Cette fois, il existe plusieurs classes d'équivalence dont la taille, c'est-à-dire le nombre de blocs de base qui les constituent, varie. Il est important de noter que dans une perspective de sécurité, il est recommandé d'avoir des classes d'équivalence dont la taille est la plus petite possible [Burow et al. 2017]. En effet, cela permet de limiter le nombre de blocs de base atteignables depuis un branchement indirect.

Une fois les cibles des branchements indirects identifiées, un mécanisme est mis en place pour contrôler la validité des cibles lors de l'exécution. On distingue les techniques qui vérifient la légitimité du couple (branchement indirect, adresse de destination) lors de l'exécution [Abadi et al. 2009] et les techniques qui remplacent les branchements indirects par de nouvelles séquences d'instructions équivalentes composées de suites de test et de branchements directs [Arthur et al. 2015].

2.3 Attaques par injection de fautes

La figure 2.1 présente le fonctionnement d'une attaque par injection de fautes. En bas de la figure, l'attaquant injecte une perturbation dans le circuit cible. Cette perturbation se traduit par une faute dans la logique du circuit et dans le fonctionnement de la microarchitecture. Ainsi, il est possible de fauter le programme exécuté par ce circuit. Un attaquant pourra exploiter l'effet de cette faute sur le programme pour compromettre une des propriétés de sécurité présentées précédemment. Bien que dans cette thèse, nous nous concentrons sur les circuits de type processeur, ces attaques fonctionnent également sur d'autres types de circuit, par exemple les accélérateurs cryptographiques.

2.3.1 Mécanismes d'injection de fautes

Les mécanismes d'injection de fautes permettent à un attaquant de soumettre le circuit cible à une perturbation physique. Les mécanismes d'injection de fautes se différencient par :

- leur précision spatiale, c'est-à-dire la capacité à sélectionner précisément un ou plusieurs bits ainsi que l'effet de la faute sur ces bits (inversion, mise à 1, mise à 0, transformation aléatoire) ;
- leur précision temporelle, c'est-à-dire la capacité à contrôler le moment d'injection, par exemple lors de l'exécution d'une instruction ;

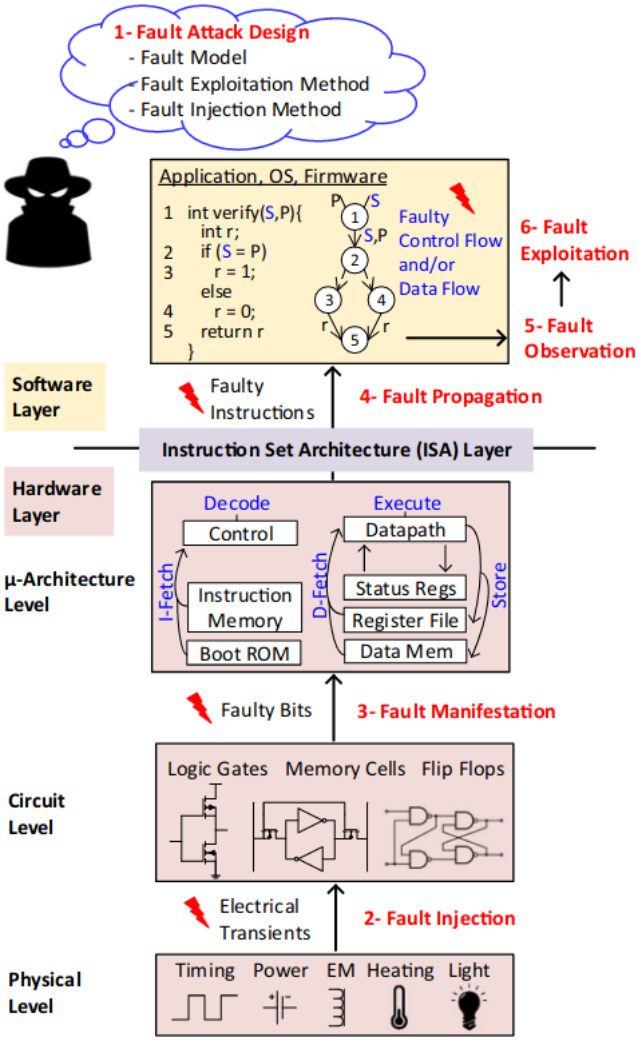

FIGURE 2.1 – Illustration du processus d'injection de faute dans un circuit intégré [YUCE et al. 2018]

— leur permanence, c'est-à-dire la durée de la perturbation ou de son effet dans le temps (un cycle, plusieurs cycles ou permanente).

Lors de la construction d'une attaque, on choisit le mécanisme d'injection de fautes en fonction de ces critères et de l'attaque visée. Certaines attaques peuvent être réalisées avec un faible niveau de précision tandis que d'autres requièrent une précision maximale pour éviter de déclencher une contre-mesure.

Perturbation de l'horloge

L'horloge d'un circuit intégré permet de cadencer les éléments synchrones de ce circuit. Lors des fronts d'horloge (montant et/ou descendant), les éléments synchrones changent d'état en fonction de la valeur de leurs entrées. La valeur minimale de la période d'horloge d'un circuit est définie par le temps maximum requis pour qu'un signal se propage dans la logique combinatoire entre deux éléments synchrones. En dessous de cette valeur, il est possible que les signaux en entrée des éléments synchrones ne soient pas stables et donc qu'une faute en sortie des éléments synchrones apparaisse.

Agoyan et al. utilisent des glitches sur le signal d'horloge pour perturber le calcul de l'algorithme AES sur un FGPA et ensuite faire une attaque par cryptanalyse du résultat erroné [Agoyan et al. 2010]. Ils montrent notamment le rapport entre la durée de la perturbation du signal d'horloge et le nombre de bits fautés dans le résultat. Korak et al. combinent les glitches sur le signal d'horloge avec des glitches sur le bus d'alimentation pour améliorer le taux de succès de l'injection de fautes [Korak et al. 2014]. Ils rapportent que la durée du glitch permet de cibler différents étages dans la microarchitecture des microcontrôleurs cibles. Claudepierre et al. proposent un mécanisme d'injection de fautes multiples à bas coût en utilisant des perturbations du signal d'horloge [Claudepierre et al. 2021]. Enfin, il existe aussi des systèmes d'injection de fautes à faible coût, commercialisés pour le grand public, tels que la carte ChipWhisperer [1].

Les mécanismes de perturbation du signal d'horloge permettent d'atteindre une précision spatiale modérée. En effet, il n'est pas possible de contrôler la valeur de la faute, mais il est possible de fauter un unique bit [Agoyan et al. 2010]. La précision temporelle est limitée à la période d'horloge dans laquelle la perturbation est injectée. Ce type de perturbation est non-permanent et le circuit reprend son fonctionnement nominal au cycle suivant. Cependant, l'effet de la perturbation peut se propager sur plusieurs cycles, par exemple lorsque la perturbation modifie la valeur contenue dans un registre.

Perturbation de l'alimentation

La vitesse de propagation des signaux électriques dans la logique combinatoire d'un circuit dépend de la tension d'alimentation de ce circuit. Ainsi, diminuer la tension d'alimentation augmente ce temps de propagation. Il peut alors apparaitre des erreurs d'échantillonnage similaires à celles observées en présence d'une perturbation sur l'horloge. Zussa et al. montrent dans leurs travaux l'équivalence entre les attaques sur l'horloge et celles sur l'alimentation [Zussa et al. 2013].

1. `https://www.newae.com/chipwhisperer`

Les techniques classiques de perturbation de l'alimentation sous-alimentent le circuit pendant une courte période [Zussa et al. 2013 ; Korak et al. 2014]. D'autres travaux proposent de créer un court-circuit pour perturber le réseau d'alimentation du circuit cible [O'Flynn 2016].

Les mécanismes de perturbation sur l'alimentation sont similaires aux mécanismes de perturbation de l'horloge. Enfin, il est important de noter que ces mécanismes permettent d'injecter de multiples fautes, par exemple pour attaquer un processus de démarrage sécurisé embarqué [Van den Herrewegen et al. 2020].

Perturbation de la température

Bien que peu courant, il est possible de perturber un circuit en modifiant sa température. Hutter et al. chauffent un microcontrôleur au-delà de la température de fonctionnement et parviennent à attaquer une implémentation logicielle de RSA [Hutter et al. 2013]. Il faut toutefois noter que les précisions temporelle et spatiale sont faibles. En effet, il est difficile de chauffer un circuit localement. Il est aussi difficile de contrôler des variations de température sur des intervalles de temps du même ordre de grandeur que la période d'horloge. Pour les mêmes raisons, il est également difficile de limiter le comportement fauté à un cycle d'horloge. La perturbation de la température n'est donc pas un mécanisme d'injection très pertinent pour un attaquant.

Perturbation électromagnétique

Émettre un champ électromagnétique proche d'un circuit permet d'induire des courants dans ce circuit ce qui génère des fautes. Dumont et al. proposent un modèle de ce phénomène [Dumont et al. 2021]. Lors d'une impulsion électromagnétique, le réseau d'alimentation du circuit est perturbé pendant la durée de l'impulsion. Cette perturbation affecte particulièrement les éléments synchrones du circuit. De plus, ils rapportent que la polarisation de l'impulsion électromagnétique permet de contrôler la nature de la faute (mise à 0 ou à 1) et que la propagation de la perturbation reste localisée autour de la sonde d'injection.

Les perturbations par impulsion électromagnétique sont populaires et font l'objet de nombreuses études [Gandolfi et al. 2001 ; Moro et al. 2013 ; Ordas et al. 2017 ; Trouchkine et al. 2021 ; Proy et al. 2019 ; Dumont et al. 2021]. Cette technique permet d'atteindre une très bonne précision spatiale et temporelle avec un effet localisé. La complexité de mise en œuvre et le coût du matériel sont toutefois légèrement supérieurs à ceux des mécanismes de perturbations de l'alimentation et de l'horloge.

Perturbation lumineuse

Un rayonnement lumineux focalisé et plus particulièrement un rayonnement laser permet de créer une faute dans un circuit intégré [Skorobogatov et al. 2003]. Il est cependant nécessaire de mettre à nu les couches de silicium du circuit en le décapsulant par abrasion mécanique ou chimique, ce qui impacte fortement la compléxité et le coût d'une attaque.

Roscian et al. proposent un modèle des perturbations lumineuses utilisant l'effet photoélectrique [Roscian et al. 2013]. Lors du passage d'un photon à travers les couches de silicium d'un circuit, un courant électrique transitoire apparait et peut perturber le fonctionnement des

transistors. Ils rapportent que l'effet d'une faute injectée par un rayonnement laser est soit la mise à 0 soit la mise à 1, mais rarement une inversion de la valeur du bit ciblé. De plus, ce type de mécanisme permet d'obtenir des fautes ciblant un unique bit très précisément [Colombier et al. 2019]. Aujourd'hui, les capacités des mécanismes d'injection laser permettent de faire des attaques avec de multiples fautes. Colombier et al. utilisent un banc laser avec quatre spots pour fauter jusqu'à 4 bits non contigus dans un même cycle ou plusieurs bits non contigus sur plusieurs cycles [Colombier et al. 2022]. Ce mécanisme d'injection de fautes permet donc de construire des attaques bien plus complexes, potentiellement capables de déjouer de nombreuses contre-mesures.

Les perturbations lumineuses permettent d'atteindre une précision spatiale et temporelle encore plus grande que les perturbations électromagnétiques. Ceux sont également des mécanismes populaires [Skorobogatov et al. 2003 ; Roscian et al. 2013 ; Dutertre et al. 2014 ; Vasselle et al. 2017 ; Guillen et al. 2017 ; Colombier et al. 2019 ; Colombier et al. 2022] bien que le coût de l'équipement et la complexité de préparation soient élevés.

Perturbation par rayon X

En 2017, Anceau et al. ont proposé une approche permettant de modifier le comportement d'un transistor dans la mémoire d'un circuit à l'aide de faisceaux focalisés de rayon X [Anceau et al. 2017]. La perturbation est cette fois semi-permanente et il est nécessaire de chauffer le circuit pour la faire disparaitre. Cela s'oppose aux mécanismes d'injection présentés précédemment pour lesquels la perturbation disparait par elle-même après quelques cycles d'horloge. Ce type d'attaque ouvre de nouvelles perspectives qui sont tout de même limitées par le coût d'accès au matériel nécessaire (ici un synchrotron).

Exploitation des fonctionnalités du circuit cible

Jusqu'ici nous avons uniquement présenté des mécanismes d'injection de fautes externes pour lesquels un accès physique au circuit cible est nécessaire. Il existe aussi des mécanismes d'injection de fautes qui utilisent des fonctionnalités internes du circuit cible et qui ne nécessitent aucun accès physique. Le prérequis commun à tous ces mécanismes est un accès logique à la cible, par exemple une connexion réseau. La perturbation physique est alors injectée par le circuit cible lui-même en détournant des propriétés physiques ou des fonctionnalités déjà présentes et contrôlables par l'attaquant.

Les attaques de type *RowHammer* permettent de modifier la valeur de bits stockés dans une cellule mémoire de type DRAM en réalisant des accès répétés aux cellules adjacentes [Y. Kim et al. 2014]. Un attaquant exécutant un programme malicieux sur la même plateforme physique qu'un programme cible peut alors modifier le contenu de la mémoire du programme cible sans y accéder. L'attaquant n'a pas besoin d'avoir un niveau de privilège particulier autre que celui de pouvoir exécuter le programme malicieux.

Il est également possible d'attaquer un système en modifiant la configuration de certains modules internes au circuit. Gravellier et al. proposent de modifier la configuration du délai de transaction mémoire dans le contrôleur mémoire pour introduire des fautes lors des

Tableau 2.1 – Comparaison des mécanismes d'injections de faute

Mécanismes d'injection	Précision spatiale	Précision temporelle	Permanence	Accès	Coût
Horloge	Modérée	Modérée	1 cycle	Physique	Faible
Alimentation	Modérée	Modérée	1+ cycles	Physique	Faible
Température	Faible	Faible	cycles	Physique	Faible
Électromagnétique	Élevée	Élevée	1+ cycles	Physique	Modéré
Laser	Élevée	Élevée	1+ cycles	Physique	Élevé
Rayon-X	Complète	-	Semi Permanent	Physique	Très élevé
Logique	Faible	Faible	1+ cycles	Logique	Très Faible

transactions ultérieures [Gravellier et al. 2021]. Plusieurs travaux proposent de modifier la configuration des périphériques de gestion de l'énergie pour injecter des perturbations sur l'alimentation ou l'horloge à distance [Tang et al. 2017 ; Qiu et al. 2019 ; Murdock et al. 2020]. Cette fois, l'attaquant a besoin d'un haut niveau de privilège pour modifier la configuration des différents modules et ainsi réaliser l'attaque. Ces attaques restent néanmoins intéressantes puisqu'elles permettent d'attaquer l'exécution d'un programme dans une enclave sécurisée telle que ARM TrustZone [Tang et al. 2017 ; Qiu et al. 2019] ou Intel SGX [Murdock et al. 2020].

Enfin, pour les circuits de type FPGA partagés entre plusieurs utilisateurs, il est possible pour un attaquant de configurer sa propre part du FPGA pour implémenter un circuit capable d'injecter des perturbations sur l'alimentation [Glamocanin et al. 2021].

2.3.2 De l'effet des fautes à leur modélisation

Une fois la perturbation injectée dans le circuit cible, elle produit une faute dans la microarchitecture. On parle alors de l'effet de la faute, et pour le caractériser on utilise souvent les mêmes critères que pour les mécanismes d'injection de fautes : précision spatiale, précision temporelle et permanence. Toutefois, l'effet de la faute peut différer de celui de la perturbation. Par exemple, une perturbation transitoire dans la mémoire peut être capturée par le cache. Pour les cycles suivants, la valeur en mémoire sera correcte, mais la valeur enregistrée dans le cache restera erronée jusqu'à ce que le cache recharge cette valeur depuis la mémoire. Dans ce cas une perturbation transitoire provoque une faute qui dure plusieurs cycles d'horloge.

Pour discuter de l'effet des fautes, il est commun d'utiliser des modèles de faute. Un modèle de faute peut être utilisé aussi bien pour caractériser un mécanisme d'injection de fautes ou pour construire une attaque que pour concevoir et évaluer une contre-mesure. On trouve dans la littérature trois grands niveaux d'abstractions pour les modèles de fautes comme présentés sur la figure 2.1 :

— Microarchitecture, par exemple une corruption de lecture de la mémoire de données ou d'instructions, ou une corruption d'un signal dans un étage du pipeline ;
— Jeu d'instructions, par exemple une corruption d'instruction ou d'un registre ;

— Code source, par exemple la corruption d'une variable de test.

Une abstraction au niveau de la microarchitecture permet de représenter l'effet et la propagation des fautes dans la cible. Cependant, la complexité de ce modèle dépend également de la complexité de la microarchitecture du circuit cible. Plusieurs travaux de caractérisation cherchent à construire des modèles de fautes au niveau de la microarchitecture [Korak et al. 2014 ; Trouchkine et al. 2021]. On trouve aussi des travaux qui utilisent des modèles de la microarchitecture du processeur pour évaluer des contre-mesures [Grycel et al. 2021].

Une abstraction au niveau du jeu d'instructions permet une plus grande expressivité de l'effet de la faute au niveau logiciel. Il permet de représenter un grand nombre de fautes proches du matériel, notamment sur l'encodage des instructions sans se préoccuper des détails d'implémentation. On retrouve au niveau du jeu d'instructions les modèles de type saut d'instructions, rejeu d'instructions, corruption de registre ou encore inversion de test. Les modèles au niveau du jeu d'instructions sont aussi bien utilisés pour de la caractérisation [Moro et al. 2013 ; Proy et al. 2019 ; Menu et al. 2020], que pour la construction de contre-mesures [Moro 2014 ; Barry 2017 ; Proy et al. 2017].

Enfin, il est possible de modéliser l'effet des fautes au niveau du code source. Cela permet de mieux identifier les effets sur le programme. Un tel niveau est donc souvent utilisé pour construire des attaques [Boneh et al. 1997 ; Biham et al. 1997 ; Piret et al. 2003]. On trouve aussi quelques travaux qui utilisent un modèle de fautes au niveau du code source, avec par exemple des inversions de condition ou des sauts de lignes de code, pour construire des contre-mesures [Lalande et al. 2014].

Les travaux de Yuce et al. utilisent une méthodologie prenant en compte des aspects de la microarchitecture, comme le pipeline et le chemin critique de chaque étage, pour construire des attaques par injection de fautes [Yuce et al. 2017]. En particulier, ils montrent que certaines contre-mesures logicielles utilisant de la duplication d'instructions sont vulnérables en exploitant la microarchitecture.

Laurent et al. étudient le comportement de la microarchitecture face aux injections de fautes [Laurent et al. 2019]. Ils montrent qu'une analyse au niveau du jeu d'instructions n'est pas suffisante pour comprendre comment un processeur répond à une attaque. Par exemple, certaines fautes dépendent du contexte logiciel notamment pour les mécanismes de *forwarding* et d'exécution spéculative. En effet, une faute sur le mécanisme de forwarding responsable de la gestion des dépendances de données entre instructions peut apparaitre comme un remplacement d'instruction au niveau du jeu d'instructions. Cependant, cette faute peut avoir différents effets sur le programme en fonction des instructions précédemment exécutées et des suivantes.

Ces travaux montrent l'importance de la prise en compte de la microarchitecture dans l'étude des attaques par injection de fautes. En effet, même si les propriétés d'intégrité du code, du flot de contrôle et des données sont assurées, une faute dans la microarchitecture peut tout aussi bien modifier le comportement du programme. Il est donc important de prendre en compte la microarchitecture lors de la conception de contre-mesure pour assurer l'exécution correcte des instructions.

2.3.3 Exploitations

La finalité d'une attaque informatique est de compromettre une des propriétés de confidentialité, d'intégrité ou d'authenticité présentées dans la section 2.2. Pour les attaques par injection de fautes, on distingue les attaques sur la confidentialité qui visent le plus souvent les systèmes cryptographiques et les attaques génériques qui visent l'intégrité ou l'authenticité.

Attaques cryptographiques

En 1997, Boneh et al. présentent l'attaque connue sous le nom de *Bellcore*, considérée aujourd'hui comme la première attaque par injection de fautes [Boneh et al. 1997]. Cette attaque vise l'implémentation de l'algorithme RSA utilisant le théorème des restes chinois. L'objectif est de fauter le résultat de l'algorithme pour le factoriser avec un résultat non fauté. Il est alors possible de retrouver la clé privée utilisée sans besoin de résoudre le problème de factorisation des grands nombres. Ce type d'attaque est appelée *Differential Fault Analysis* (DFA) par Biham et al. qui s'intéressent à l'algorithme DES [Biham et al. 1997]. Piret et al. proposent une attaque sur l'algorithme AES [Piret et al. 2003]. Les attaques DFA exploitent la différence entre un calcul fauté et un calcul correct. D'autres variantes exploitent des collisions ou l'absence d'erreur dans le résultat pour gagner en connaissance sur le secret. Clavier présentent plusieurs techniques de cryptanalyse basées sur les injections de fautes pour les algorithmes de chiffrement par bloc [Clavier 2012] alors que Berzati et al. se concentrent sur l'algorithme RSA [Berzati et al. 2012]. On peut aussi citer les attaques de type *Statistical Inefective Fault Analysis* qui consistent à injecter une faute qui ne modifie pas le résultat du calcul final [Dobraunig et al. 2018].

Il est important de noter que les protections contre les attaques par canaux auxiliaires telles que le masquage ne permettent pas de se protéger contre les attaques par injection de fautes [Amiel et al. 2006 ; Pan et al. 2019].

Enfin, les attaques par injection de fautes peuvent aussi être un outil pour la rétro ingénierie [Biham et al. 1997 ; San Pedro et al. 2011].

Attaques génériques

Bien qu'une grande partie de la sécurité informatique repose sur la cryptographie, il est aussi nécessaire de construire des protocoles pour mettre en place les différents dispositifs de sécurité. Il est parfois plus simple pour un attaquant de cibler la logique du protocole ou de l'implémentation de la fonction cryptographique que la fonction cryptographique elle-même. Souvent les attaques informatiques essaient de corrompre l'intégrité du code ou du flot de contrôle.

Une cible courante des attaques par injection de fautes sont les systèmes de démarrage sécurisé. Timmers et al. présentent une attaque permettant d'exécuter du code arbitrairement lors du démarrage sécurisé d'un SoC ARM en utilisant une perturbation sur l'alimentation [Timmers et al. 2016]. Cette attaque transforme une instruction de chargement depuis la mémoire en une instruction de branchement qui saute dans une section de la mémoire contrôlée par l'attaquant. Vasselle et al. attaquent le démarrage sécurisé d'un smartphone Android avec des perturbations laser [Vasselle et al. 2017]. Plutôt que d'attaquer directement le programme,

ils modifient le contenu du registre de status (CPSR) qui contient entre autres le résultat d'une comparaison et le registre de configuration du niveau de privilège (CSR). Van den Herrewegen et al. attaquent le démarrage sécurisé d'un microcontrôleur en combinant une double faute sur l'alimentation et la technique du *Return Oriented Programming* [Van den Herrewegen et al. 2020]. On retrouve même sur des forums l'utilisation d'injection de fautes pour contourner le démarrage sécurisé de la console de jeux vidéo Xbox 360 [GliGli et al. 2012]. Cela confirme que les attaques par injection de fautes sont une menace réelle et que le profil des attaquants n'est pas limité à des experts dans des laboratoires spécialisés.

Ces attaques peuvent être adaptées à un grand nombre de cas d'utilisation, par exemple l'élévation de privilège sur un système Linux en utilisant cette fois une corruption des données [Seaborn et al. 2015]. On observe qu'il existe plusieurs attaques ciblant soit l'intégrité du code et du flot de contrôle, soit l'intégrité des données. Il est donc important d'assurer toutes ces propriétés à la fois pour couvrir tous les chemins d'attaques possibles.

2.4 Protections

Dans la section précédente, nous avons montré le besoin de se protéger contre les attaques par injection de fautes. Il est possible d'implémenter une protection au niveau analogique dans le circuit, au niveau logique ou au niveau du logiciel.

Plusieurs travaux proposent des capteurs pour détecter les perturbations physiques dans le circuit. Ainsi, il est possible d'utiliser des détecteurs de glitches basés sur la mesure de la période d'horloge [Endo et al. 2012], des détecteurs d'impulsions électromagnétiques [El-Baze et al. 2016] ou des détecteurs d'illumination laser [He et al. 2016]. Une autre solution est de placer le circuit dans un boîtier robuste aux attaques par injection de fautes [Sohier et al. 2022]. D'autres travaux proposent des modifications lors de la conception des circuits pour les rendre plus robustes face aux fautes [Karaklajić et al. 2013]. L'intégrité des données et du code en mémoire peut être assurée par des codes détecteurs d'erreur tels que les CRC. Les unités de calcul sont souvent protégées en utilisant le principe de redondance. Par exemple, il est possible de réaliser deux fois le même calcul en parallèle et de comparer les deux résultats. Néanmoins, cette solution double la surface du circuit nécessaire pour réaliser le calcul. Une autre solution est d'utiliser de la redondance temporelle en calculant plusieurs fois le même résultat avec la même unité logique.

Dans cette section, nous nous intéressons aux protections placées aux niveaux logiciel et mixte, c'est-à-dire avec une modification conjointe du matériel et du logiciel.

2.4.1 Protections logicielles

Afin de protéger un programme contre les attaques par injection de fautes, il est possible de modifier uniquement le logiciel. Il existe plusieurs familles de contre-mesures purement logicielles.

Une partie des contre-mesures logicielles se concentre sur la détection de faute sur le flot de contrôle. Plusieurs contre-mesures utilisent des compteurs pour suivre l'évolution du flot de contrôle lors de l'exécution. Les compteurs sont initialisés avec une valeur unique avant le début

de la séquence d'instructions à vérifier. Ils sont ensuite incrémentés en fonction du chemin de flot de contrôle pris durant l'exécution. En fin de séquence, les compteurs sont comparés à des valeurs de référence. Une divergence entre la valeur du compteur et la valeur de référence permet de détecter une faute sur le flot de contrôle. Lalande et al. proposent d'incrémenter le compteur entre chaque ligne du code source à protéger [Lalande et al. 2014]. Barry proposent plutôt d'incrémenter le compteur entre chaque instruction assembleur. Bien que très coûteuse, car doublant la taille de code, cette technique vérifie également que le bon nombre d'instructions a été exécuté dans la séquence protégée [Barry 2017].

Quelques contre-mesures proposent des solutions pour protéger les données liées au flot de contrôle. Schilling et al. proposent de protéger le calcul des branchements conditionnels [Schilling et al. 2018]. Les variables entières utilisées pour le calcul des conditions sont encodées avec un AN-code. Les AN-codes sont des codes détecteurs d'erreurs linéaires par rapport à l'addition qui est l'opération utilisée pour le calcul de condition. Le résultat de la comparaison est obtenu par le calcul du modulo de la soustraction entre les deux opérandes à comparer. Les auteurs discutent également de la combinaison du résultat de la comparaison avec une contre-mesure protégeant l'intégrité du flot de contrôle.

Proy et al. se concentrent sur la protection automatique des boucles en dupliquant les instructions et les données liées au flot de contrôle des boucles lors de la compilation [Proy et al. 2017]. Ils proposent aussi de sécuriser les appels de fonction en utilisant des variables de suivies lors des appels et des retours de fonction. Il est ainsi possible de vérifier le nombre de fois qu'une fonction a été appelée dans un programme.

SWIFT est une protection logicielle contre les *Single Event Upset* [Reis et al. 2005]. Cette protection fait l'hypothèse que la mémoire est déjà protégée par un mécanisme de code correcteur d'erreurs. Elle combine un mécanisme de redondance basé sur la duplication d'instruction et un mécanisme de flot de contrôle qui associe une signature unique à chaque bloc de base. Ces travaux présentent également un ensemble d'optimisations permettant de réduire le coût de la contre-mesure. En comparaison avec les autres protections, SWIFT assure en même temps l'intégrité du code, du flot de contrôle et des données. De plus, les transformations de code présentées dans ces travaux peuvent être implémentées dans un compilateur pour une application automatique de la contre-mesure.

Lorsque la disponibilité devient une propriété essentielle d'un système, il n'est plus suffisant de détecter les fautes. Plusieurs travaux proposent des solutions de tolérance aux fautes qui permettent d'assurer le bon comportement du programme même en présence de fautes. Ces contre-mesures sont souvent construites pour résister à un modèle de faute de type saut d'instruction. Une technique classique est d'utiliser de la redondance temporelle ou spatiale. Barenghi et al. proposent de tripler les instructions en stockant les résultats dans des registres différents [Barenghi et al. 2010]. Il est alors possible de détecter et de corriger une faute en conservant le résultat majoritaire. À cela, Barenghi et al. proposent aussi d'utiliser un bit de parité pour valider l'intégrité des données chargées depuis la mémoire.

Moro propose une contre-mesure de tolérance au saut d'une instruction [Moro 2014]. Pour cela, les instructions doivent être idempotentes ou transformées dans une forme idempotente équivalente, c'est-à-dire que les instructions produisent le même résultat qu'elles soient exécutées une ou plusieurs fois. Après cette transformation, les instructions sont dupliquées

permettant ainsi de résister à une attaque qui sauterait une instruction. Barry montre qu'il est possible de déployer cette contre-mesure automatiquement lors de la compilation [Barry 2017]. De plus, il propose une généralisation de ce mécanisme avec de la n-plication d'instruction pour résister à un saut de n instructions.

Plusieurs travaux proposent des contre-mesures spécifiques à un algorithme. Patrick et al. proposent une méthode de détection des fautes pour les algorithmes de chiffrement par bloc à l'aide de redondance intra-instruction [Patrick et al. 2017]. L'idée repose sur une représentation par découpage binaire (bit-slicing) de l'état de l'algorithme. L'état pouvant être représenté comme une matrice de N blocs de taille M, chaque bloc étant traité individuellement. Le découpage binaire consiste alors à utiliser la matrice transposée. Une nouvelle méthode de calcul est nécessaire afin de manipuler les données ainsi éclatées. Afin de renforcer la sécurité, plusieurs états redondants peuvent être entrelacés dans la matrice avec un état de référence. Cela permet de vérifier la cohérence entre les états redondants ainsi que de valider le bon calcul de l'état de référence.

Lac et al. présentent une contre-mesure utilisant aussi du découpage binaire [Lac et al. 2018]. Cette fois, une implémentation sur 8-bit d'un algorithme est parallélisé sur une architecture 32 bits. Il est donc possible de traiter quatre variables en parallèle grâce à des instructions SIMD. La donnée à manipuler est dupliquée et intercalée avec une donnée de référence. Il est alors possible de vérifier la cohérence entre chaque donnée de 8 bits présente deux fois dans chaque mot de 32 bits. De plus, la présence d'une donnée de référence permet de valider que le calcul s'est bien déroulé dans son intégralité.

Enfin, une dernière catégorie de contre-mesures logicielles, dites *infectives*, protègent spécifiquement les algorithmes cryptographiques. Elles ont vu le jour pour répondre à la problématique de *single point of failure* des techniques de détection souvent basées sur un test (représenté par un unique bit). Le concept est de propager un calcul fauté pour *infecter* le résultat final afin que celui-ci soit indistinguable d'une valeur aléatoire indépendante des secrets manipulés dans l'algorithme. Afin de détecter une erreur, l'algorithme à sécuriser est calculé deux fois. La différence entre les deux algorithmes est calculée à l'aide d'un *OU exclusif* puis elle est transformée par une fonction non linéaire avant d'être recombinée avec l'état de l'algorithme à protéger. Ces contre-mesures sont limitées à des applications cryptographiques et ne protègent que contre certains types d'attaquant. En effet, une double faute permet de déjouer ces contre-mesures en attaquant les deux calculs redondants. De plus les techniques de type SIFA exploitent une faute qui ne modifie pas le résultat et sont donc insensibles à ces contre-mesures [Baksi et al. 2019].

Bien que les contre-mesures logicielles permettent de répondre à certaines attaques par injection de fautes, les propriétés de sécurité qu'elles assurent sont limitées. Les contre-mesures logicielles pures ne permettent pas de garantir l'intégrité du code pendant l'exécution. Au mieux, il est possible de vérifier l'intégrité du code lors du chargement de celui-ci en mémoire par un système de démarrage sécurisé. De plus, le coût sur les performances du système à protéger est important en ce qui concerne la mémoire nécessaire et le temps d'exécution. Le principe de duplication, par exemple, double à la fois l'espace mémoire nécessaire et le temps d'exécution pour les sections protégées. Pour toutes ces raisons, de plus en plus de travaux

étudient les contre-mesures mixtes matérielles et logicielles. Pour autant, les contre-mesures logicielles restent intéressantes lorsque le système à protéger est déjà déployé et qu'il n'est plus possible de modifier le matériel, car elles sont plus fléxibles.

2.4.2 Protections mixtes matérielles et logicielles

Combiner les approches matérielles et logicielles permet de concevoir des contre-mesures dont le coût en temps d'exécution et en taille mémoire est plus faible que pour les contre-mesures logicielles pures. De plus, l'intégration des contre-mesures directement dans le matériel peut augmenter le niveau de sécurité apporté notamment en utilisant des informations qui ne sont pas accessibles facilement au niveau logiciel, par exemple la valeur de l'instruction effectivement chargée dans le pipeline. On distingue les contre-mesures qui ajoutent un co-processeur au support matériel existant et les contre-mesures qui modifient directement le support matériel en s'intégrant dans la microarchitecture du processeur. Cette section se concentre plus particulièrement sur les contre-mesures assurant les propriétés d'intégrité du code et du flot de contrôle, car cette thèse vise à protéger le code et son exécution.

Pour assurer l'intégrité du code, une technique commune aux contre-mesures mixtes matérielles logicielles est le calcul d'une signature représentant les séquences d'instructions valides selon le programme qui s'exécute. Le calcul de la signature lors de l'exécution est réalisé par le matériel, ce qui permet de limiter la dégradation des performances du programme en temps d'exécution au prix d'une augmentation de la surface du circuit. Il est toutefois nécessaire d'ajouter des métadonnées et du code au programme à protéger afin de comparer les signatures calculées lors de l'exécution à des signatures de référence préalablement calculées sur le code binaire du programme. Plusieurs contre-mesures reposent sur un tel calcul de signature sur des blocs d'instructions et vérifient ainsi l'intégrité du code par partie [Arora et al. 2006 ; Danger et al. 2020 ; Clercq et al. 2016]. Ces mécanismes de signature peuvent être combinés avec des mécanismes d'intégrité du flot de contrôle. Par exemple, une machine à états permet de valider l'enchainement des séquences d'instructions à partir de métadonnées embarquées dans le programme. Arora et al. utilisent plusieurs machines à états placées dans un co-processeur pour valider le graphe d'appel et l'enchainement des instructions à l'intérieur d'une fonction [Arora et al. 2006]. Danger et al. utilisent un co-processeur qui lit une mémoire dédiée dans laquelle sont stockées entre autres la signature d'intégrité du code ainsi que la liste des adresses des blocs successeurs valides [Danger et al. 2020].

La *General Path Signature Analysis* (GPSA), une technique originellement proposée par Wilken et al. dans le cadre de la sûreté de fonctionnement en présence de *Single Event Upset*, permet d'assurer l'intégrité du code et l'intégrité du flot de contrôle avec le calcul d'une seule signature pour les deux propriétés de sécurité [Wilken et al. 1990]. Werner et al. proposent d'utiliser ce mécanisme dans le cadre des attaques par injection de fautes [Werner et al. 2015]. L'idée est de calculer une signature à partir des instructions d'un bloc de base comme pour les mécanismes d'intégrité du code. La signature est chaînée entre chaque bloc de base, ce qui permet également d'assurer l'intégrité du flot de contrôle. Pour cela, il est nécessaire d'instrumenter le code pour garantir des invariants dans le flot de contrôle. Cette technique est présentée en détail dans le chapitre 3.

Les contre-mesures [Arora et al. 2006 ; Werner et al. 2015 ; Danger et al. 2020] utilisent

toutes les trois des co-processeurs pour le calcul de la signature. Cette technique a l'avantage d'être applicable sans modifier la microarchitecture du processeur à protéger.

Il existe aussi des contre-mesures qui s'intègrent dans la microarchitecture du processeur. SOFIA est une architecture de processeur sécurisé qui assure la confidentialité et l'authenticité du code ainsi que l'intégrité flot de contrôle. La confidentialité du code et l'intégrité du flot de contrôle sont assurées par un même mécanisme. Le code est chiffré en mémoire et SOFIA repose sur une dépendance entre le déchiffrement des instructions et le flot de contrôle. Ainsi, lors du chargement d'une instruction dans le processeur, le déchiffrement dépend de l'adresse de l'instruction courante ainsi que de l'adresse de l'instruction précédente. Une modification du code ou du flot de contrôle provoque donc un mauvais déchiffrement de l'instruction. Un second mécanisme fonctionnant sur le principe de signature, présenté précédemment, assure l'authenticité (et donc l'intégrité aussi) du code. Pour l'authenticité du code, SOFIA utilise une primitive cryptographique pour construire un *Message Authentication Code* (MAC). Cette solution permet de détecter un éventuel déchiffrement fauté et donc toutes les fautes ciblant le code ou le flot de contrôle. Bien que très intéressante pour les propriétés de sécurité assurées, cette contre-mesure possède un coût important. D'une part, il est nécessaire d'utiliser deux primitives cryptographiques, une pour le déchiffrement et une pour le calcul du MAC, ce qui entraine une augmentation de la surface du circuit de 28,2 %. D'autre part, les modifications nécessaires au niveau logiciel provoquent un surcoût de 140 % de la taille de code.

Werner et al. proposent une amélioration de SOFIA qui combine GPSA et la construction de fonctions de chiffrement authentifié à partir d'une fonction éponge [Werner et al. 2018]. Cette fois, une unique fonction de chiffrement authentifié assure l'authenticité du code et l'intégrité du flot de contrôle, ainsi que la confidentialité du code. Un mécanisme similaire à GPSA permet de créer des invariants dans le flot de contrôle sur lesquels repose le bon déchiffrement des instructions. De plus, il n'est pas nécessaire de vérifier une signature lors de l'exécution, puisque le mauvais déchiffrement d'une instruction provoque une réaction en chaîne détectée par l'apparition d'instructions invalides (n'appartenant pas au jeu d'instructions) dans le processeur. Toutefois, cela suppose que le jeu d'instructions n'est pas dense (il existe des encodages d'instruction invalides). Cette technique nécessite ainsi des métadonnées que pour les invariants de flot de contrôle, ce qui la rend particulièrement légère en comparaison des autres contre-mesures protégeant l'intégrité du code et du flot de contrôle. Cette contre-mesure entraine un surcoût moyen en taille de code et en temps d'exécution de 19.8% et 9.1% respectivement et une augmentation de la surface du circuit de 47 %.

Confidaent est une autre contre-mesure qui combine chiffrement de code et intégrité du code et du flot de contrôle [Savry et al. 2020]. Une fonction de chiffrement authentifié assure la confidentialité et l'authenticité du code en mémoire. L'intégrité du flot de contrôle est assurée par un mécanisme de masquage similaire à GPSA. Pour chaque instruction, un masque est calculé à partir du masque précédent. Ce masque est utilisé pour démasquer l'instruction courante avant son exécution. Une faute sur le flot de contrôle provoque une divergence dans le calcul des masques et donc un mauvais démasquage des instructions. Comme pour [Werner et al. 2018], la faute est détectée par l'apparition d'instructions invalides dans le pipeline du processeur. On peut ajouter que cette contre-mesure permet d'assurer la confidentialité et l'authenticité des données. Cependant, l'utilisation d'une primitive de chiffrement authentifié sur le bus mémoire impacte fortement les performances notamment en ce qui concerne la fréquence de

fonctionnement du circuit, ainsi la fréquence du processeur est 10 fois inférieure à la fréquence de l'unité de calcul de chiffrement authentifié.

Les contre-mesures mixtes matérielles et logicielles permettent d'atteindre un meilleur niveau de sécurité que les contre-mesures logicielles. Néanmoins, ces contre-mesures imposent une modification du support matériel parfois dans la microarchitecture. Cela impose l'intégration des contre-mesures tôt dans le processus de conception des systèmes. De plus, ces contre-mesures ne sont pas toujours plus légères que les solutions logicielles pures. Il est toutefois possible de réduire grandement le surcoût logiciel tout en maintenant un surcoût matériel faible comme le montre la contre-mesure [Werner et al. 2018]. Enfin, même si ces contre-mesures sont intégrées directement au niveau matériel, elles se concentrent toutes sur la protection de l'intégrité du code et du flot de contrôle à l'entrée du processeur. En conséquence, elles ne protègent pas l'exécution des instructions une fois celles-ci dans le processeur, ce qui ne couvrent pas toutes les vulnérabilités possibles en cas d'injection de fautes comme expliqué en section 2.3.2.

2.5 Conclusion

Les attaques par injection de fautes permettent aujourd'hui d'attaquer aussi bien des applications cryptographiques telles que les algorithmes de chiffrement, que les applications génériques telles que le démarrage sécurisé. Pour cela, un attaquant dispose de plusieurs mécanismes d'injection de fautes. Les mécanismes d'injection de fautes les plus simples tels que les perturbations sur l'alimentation permettent de construire des attaques complexes notamment à l'aide de fautes multiples [Van den Herrewegen et al. 2020]. Il existe aussi des mécanismes d'injection de fautes plus coûteux comme les perturbations laser avec lesquelles il est aujourd'hui possible de fauter précisément plusieurs bits non contigus avec une haute précision temporelle [Colombier et al. 2022]. Un attaquant à l'état de l'art peut donc aujourd'hui réaliser des attaques en utilisant des fautes multiples tout en conservant des précisions temporelle et spatiale élevées. En parallèle de l'étude des mécanismes d'injection de fautes, les travaux de Laurent et al. mettent en évidence la présence de vulnérabilités dans la microarchitecture des processeurs.

Aujourd'hui, il n'est plus suffisant d'assurer l'intégrité du code, du flot de contrôle et des données pour se protéger contre les fautes. L'exécution des instructions dans la microarchitecture peut aussi être impactée par une faute et donc compromettre la sécurité du système. Il est nécessaire d'assurer une autre propriété de sécurité que nous introduisons et que nous appelons *intégrité d'exécution* qui protège l'exécution des instructions dans la microarchitecture. Cette thèse vise à étudier comment protéger l'intégralité du chemin d'instructions en assurant les propriétés d'intégrité du code, du flot de contrôle et d'exécution tout en minimisant l'impact sur les performances logicielles et matérielles, et à proposer une solution viable pour un système complet.

Nous proposons les concepts d'une nouvelle contre-mesure qui répond à ces problématiques dans le chapitre 3. Nous présentons ensuite deux implémentations et une évaluation de cette contre-mesure dans le chapitre 4.

Chapitre 3

Microarchitecture sécurisée

Sommaire du présent chapitre

3.1 Introduction

Dans le chapitre 2, nous avons présenté les attaques par injection de fautes. Nous avons montré le besoin d'assurer plusieurs propriétés de sécurité : l'intégrité des données, l'intégrité du code et l'intégrité du flot de contrôle. De plus, nous avons mis en évidence le besoin de protéger la microarchitecture qui est, elle aussi, être vulnérable aux attaques par injection de fautes. Pour répondre à ce besoin, nous proposons une nouvelle propriété de sécurité dont l'objectif est de protéger l'exécution des instructions dans la microarchitecture : l'intégrité d'exécution. Dans nos travaux, nous considérons l'intégrité d'exécution comme l'intégrité des signaux de contrôle dans la microarchitecture.

Dans ce chapitre, nous présentons SCI-FI, une nouvelle contre-mesure contre les attaques par injection de fautes. Les objectifs de SCI-FI sont de couvrir l'ensemble du chemin d'instructions face aux attaques par injection de fautes en assurant les propriétés d'intégrité du code, du flot de contrôle et d'exécution. Notre premier défi est donc de proposer un mécanisme d'intégrité d'exécution pour protéger la logique de contrôle dédiée à l'exécution des instructions. Notre second défi est de combiner l'intégrité d'exécution avec l'intégrité du code et du flot de contrôle, en présence d'attaques par injection de fautes. Un dernier défi est de proposer une solution complète avec du support logiciel et matériel compatible pour une utilisation dans des systèmes embarqués tout en conservant un coût logiciel et matériel faible. SCI-FI est composée d'une extension de la microarchitecture d'un processeur ainsi que de support logiciel dans la chaîne de compilation. SCI-FI est conçue pour détecter les injections de faute simples ou multiples qui modifieraient :

— les instructions en mémoire, sur un bus ou dans le processeur ;
— le flot de contrôle ;
— le décodage des instructions ;
— un signal de contrôle lié à une instruction présente dans le pipeline.

De plus, SCI-FI supporte les branchements indirects et les interruptions pour être totalement compatible avec une utilisation par des systèmes embarqués.

La section 3.2 présente le modèle de menace et le type de systèmes considérés pour la conception de SCI-FI. La section 3.3 présente brièvement des concepts déjà connus de l'état de l'art et sur lesquels nous nous appuyons pour la construction de notre contre-mesure. La section 3.4 détaille les concepts de notre contre-mesure. Nous discutons en section 3.5 de l'utilisation de SCI-FI avec des microarchitectures de processeur plus complexes. Enfin, dans la section 3.6 nous comparons SCI-FI aux contre-mesures existantes.

Les travaux de ce chapitre ont été publiés dans la conférence *Design Automation & Test in Europe* (DATE) en 2022 [Chamelot et al. 2022]. Ils ont également été présentés dans les workshops JAIF2021 en septembre 2021 à Paris (France) [Chamelot 2021] et PHISIC2022 en mai 2022 à Gardanne (France) [Chamelot 2022].

3.2 Motivations

Avant de décrire les principes de notre contre-mesure contre les attaques par injection de fautes, cette section présente notre modèle de menace et le choix du type de processeurs ciblés.

3.2.1 Modèle de menace

Dans le chapitre 2, nous avons vu qu'il existe quatre propriétés de sécurité applicables à différents éléments d'un système. Dans le contexte des attaques par injection de fautes sur un processeur, il est possible de cibler les données conservées en mémoire ou dans des registres, ou alors le chemin d'instructions en modifiant les instructions en mémoire, leur ordre d'exécution ou leur exécution dans la microarchitecture. Dans cette section, nous considérons un attaquant

qui cherche à compromettre l'intégrité du chemin d'instructions, c'est-à-dire soit l'intégrité du code, du flot de contrôle ou d'exécution en utilisant des attaques par injection de fautes uniquement. De par leur coût de mise en œuvre, les attaques par injection de fautes sont utilisées en dernier recourt lorsque toutes les autres techniques d'attaques ont échoué. L'utilisation d'autres techniques d'attaques logicielles n'est donc pas considérée pour la conception de notre contre-mesure. Nous faisons également l'hypothèse que le stockage des données et le traitement des données dans les unités logiques du processeur sont protégés par des protections tierces. Par exemple, l'utilisation de code détecteur d'erreurs pour le stockage des données en mémoire ou dans les registres. De plus, nous ne considérons pas les problèmes liés aux autres attaques matérielles comme les attaques par canal auxiliaire ou le *microprobing*.

3.2.2 Type de processeur visé

Dans ce chapitre, nous proposons une contre-mesure pour des processeurs simples tels que les processeurs utilisés dans les cartes à puce ou les microcontrôleurs. Nous considérons un pipeline composé des cinq étages suivants :

1. *Fetch* : charge une instruction depuis la mémoire dans le pipeline ;
2. *Decode* : décode une instruction, sélectionne les opérandes dans le banc de registres et contrôle les étages suivants ;
3. *Execute* : calcule les opérations arithmétiques et logiques à partir d'une instruction décodée ;
4. *Memory* : réalise les transactions avec la mémoire de données ;
5. *Write-back* : écrit le résultat de l'instruction dans le banc de registres.

Le pipeline exécute les instructions une par une et dans l'ordre. La section 3.5 en fin de chapitre discute du potentiel d'adaptation de notre contre-mesure à des microarchitectures plus complexes.

3.3 Signature pour l'intégrité du code et du flot de contrôle

La contre-mesure que nous proposons s'appuie sur une technique de signature d'intégrité du code et du flot de contrôle déjà connue de l'état de l'art. Cette section présente le principe de fonctionnement des techniques à base de signature nécessaires à la construction de notre contre-mesure.

Dans le chapitre 2, nous avons présenté différents types de contre-mesure, notamment les contre-mesures mixtes matérielles et logicielles. Plusieurs de ces contre-mesures utilisent un mécanisme de signature pour assurer l'intégrité du code. Une signature S_i associée à un bloc de base B_i, composé des instructions $I_0, \ldots, I_n$ est calculée à l'aide d'une fonction de signature f et d'un vecteur d'initialisation IV_i.

$$s_{i_0} = f(IV_i, I_0), \quad s_{i_n} = f(s_{i_{n-1}}, I_n), \quad S_i = s_{i_n} \tag{3.1}$$

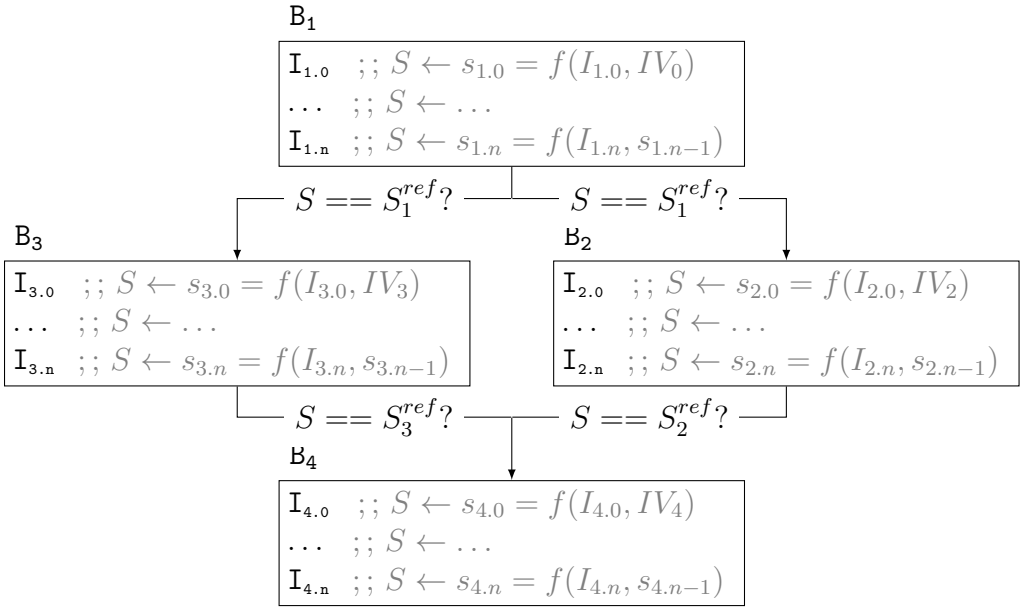

FIGURE 3.1 – Exemple du calcul d'une signature d'intégrité du code pour plusieurs blocs de base. La signature courante S est mise à jour pendant l'exécution de toutes les instructions $I_{i.j}$

La signature calculée durant l'exécution du programme est mise à jour pour chaque instruction exécutée dans un bloc de base. Cette signature est comparée régulièrement à des signatures de référence, par exemple lors de transfert de flot de contrôle. Un exemple de ce processus est présenté figure 3.1. Les signatures de référence sont pré-calculées de la même manière avant l'exécution du programme. Elles sont ensuite placées dans une mémoire dédiée ou embarquées dans le code, par exemple en plaçant les signatures de référence à la fin des blocs de base.

Comme déjà évoqué dans le chapitre 2, section 2.4.2, la technique *General Path Signature Analysis* (GPSA) utilise également une signature pour assurer l'intégrité du code et du flot de contrôle [WILKEN et al. 1990]. Pour faire le lien entre les blocs de base, et donc assurer l'intégrité du flot de contrôle, GPSA utilise la signature du bloc précédent B_{i-1} comme vecteur d'initialisation IV_i pour le bloc suivant B_i. Chaque bloc de base (et chaque instruction dans un bloc de base) est associé à une signature unique. Lorsqu'un bloc de base possède plusieurs prédécesseurs, il n'est pas possible de lui associer un IV unique, car chaque prédécesseur à une signature unique. Cependant, cela est nécessaire, notamment pour supporter les structures de contrôle telles que les boucles où l'on souhaite obtenir les mêmes signatures indépendamment du nombre d'itérations. Pour assurer un IV unique pour chaque bloc de base, GPSA utilise des ajustements de signature lorsque les blocs de base ont plusieurs prédecesseurs. L'ajustement des signatures se fait à l'aide d'une fonction de mise à jour qui est appliquée sur $N-1$ des N prédécesseurs, de telle sorte que toutes les signatures aient la même valeur. La fonction de mise à jour u modifie la signature à partir de la signature courante et d'une valeur spécifique à chaque bloc de base que nous appelons *patch*. Ainsi, pour les blocs de base B_i et B_j associés aux signatures S_i et S_j et aux patchs P_i et P_j, et ayant pour successeur le bloc de base B_k la

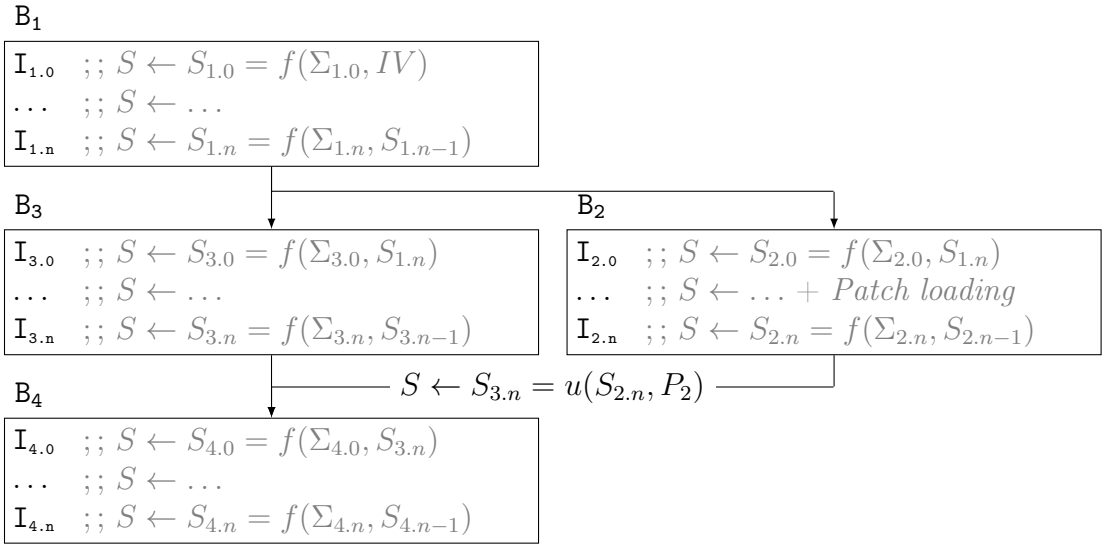

FIGURE 3.2 – Exemple du calcul d'une signature GPSA pour plusieurs blocs de base. Une mise à jour est nécessaire dans le bloc de base $\mathtt{B_2}$ à cause des blocs de base $\mathtt{B_2}$ et $\mathtt{B_3}$ qui se rejoignent.

fonction de mise à jour est utilisée suivant :

$$IV_k = u(S_i, P_i) = u(S_j, P_j) = \dots \tag{3.2}$$

Dans la figure 3.2, le bloc de base $\mathtt{B_4}$ est le successeur commun aux blocs de base $\mathtt{B_2}$ et $\mathtt{B_3}$. La signature du bloc de base $\mathtt{B_2}$ est mise à jour avec le patch P_2. De cette façon, lors de la transition de $\mathtt{B_2}$ à $\mathtt{B_4}$, la signature est modifiée pour être égale à $S_{3.n}$.

GPSA est une technique d'analyse statique, ce qui signifie que les signatures de référence et les patchs sont calculés avant l'exécution du programme. Lors de l'exécution, un support dédié à GPSA permet de calculer les signatures, charger les patchs, réaliser les mises à jour et vérifier les signatures par rapport aux signatures de référence. Pour les blocs de base nécessitant une mise à jour, il faut indiquer où trouver le patch associé et quand déclencher la mise à jour. Pour cela, il est possible d'utiliser un coprocesseur, d'étendre le jeu d'instructions ou de modifier la sémantique d'instructions déjà existantes. Il en est de même pour les vérifications des signatures pour lesquelles il faut indiquer quand réaliser une vérification et où est la signature de référence associée à cette vérification. Les patchs et les signatures de référence peuvent être conservés directement dans le code ou dans une mémoire de données dédiée. L'instrumentation du programme avec le support des mises à jour et des vérifications est souvent réalisé lors de la compilation du programme.

3.4 SCI-FI : Intégrité d'exécution

Dans cette section, nous présentons en détail les différents éléments qui composent SCI-FI.

3.4.1 Vue d'ensemble

SCI-FI articule deux modules matériels autour d'un ensemble de signaux de contrôle émis par l'étage decode. Nous appelons cet ensemble de signaux de contrôle le *pipeline state*. L'objectif de SCI-FI est d'assurer l'intégrité des signaux de contrôle qui composent le pipeline state pour assurer l'intégrité d'exécution. Pour cela, un premier module calcule une signature à partir du pipeline state en utilisant la technique GPSA, assurant également l'intégrité du code et du flot de contrôle. Un second module utilise un mécanisme de redondance pour compléter la couverture de l'intégrité d'exécution dans le pipeline. La figure 3.3 illustre un exemple d'intégration de SCI-FI dans un pipeline de processeur à 5 étages. L'étage decode émet un pipeline state Σ_i pour chaque instruction qu'il traite. En parallèle de l'étage execute, le module d'intégrité du code et du flot de contrôle (CCFI) implémente le support matériel pour GPSA et assure l'intégrité d'exécution jusqu'à la sortie de l'étage decode. Le module d'intégrité des signaux de contrôle (CSI), placé en parallèle des étages memory et write-back, complète la couverture de l'intégrité d'exécution en utilisant un mécanisme de redondance. Au niveau logiciel, le back-end du compilateur est modifié pour supporter l'instrumentation du programme requise par GPSA (mise à jour et vérification de signature).

Dans l'état de l'art, les contre-mesures qui implémentent GPSA ou une technique dérivée comme [Werner et al. 2015 ; Werner et al. 2018] utilisent l'encodage binaire de l'instruction pour calculer la signature. SCI-FI se différencie en calculant la signature dans le module CCFI à partir du pipeline state Σ_i dans la microarchitecture. Le module CSI vérifie que les signaux du pipeline state sont propagés correctement jusqu'à leur utilisation dans les étages suivants. Les signaux du pipeline state sont dupliqués dans le module CSI à la sortie de l'étage decode. Ensuite, le module CSI compare les signaux originaux aux signaux dupliqués entre chaque étage. Ainsi, le module CSI peut détecter n'importe quelle faute qui modifierait un signal du pipeline state après l'étage décode. L'intégrité de l'ensemble du chemin d'exécution est assurée par la combinaison des modules CCFI et CSI : le module CCFI assure l'intégrité du pipeline state et le module CSI assure l'intégrité de la propagation des signaux dans la microarchitecture.

Nous défendons l'architecture de notre contre-mesure composée de deux modules face à une architecture composée d'un unique module. Lors de l'exécution d'une instruction par le pipeline, de nombreux évènements dynamiques peuvent survenir. Par exemple, le gel du pipeline dû à la latence des transactions mémoire ou l'apparition d'une interruption. Ces évènements impactent la propagation des signaux dans les différents étages du pipeline. Il devient alors complexe de calculer une signature par analyse statique à partir des signaux de contrôle de tous les étages. Pour faire face à ce problème, nous proposons de calculer une signature à partir de signaux de contrôle émis par le même étage du pipeline. Notre second module permet de s'assurer que les signaux se propagent correctement. Il est possible de prendre en compte les évènements dynamiques dans le second module, car celui-ci n'utilise pas de métadonnées préalablement calculées.

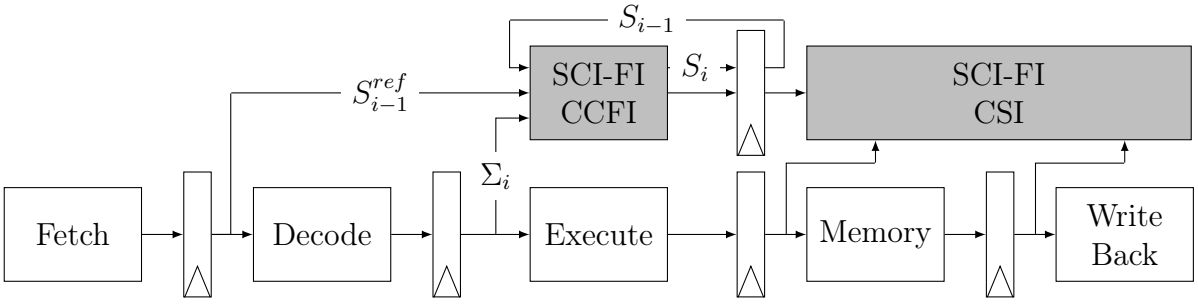

FIGURE 3.3 – Illustration d'un pipeline 5 étages étendu avec SCI-FI (modules gris)

3.4.2 Pipeline state

Le pipeline state est un vecteur de bits composé de signaux de contrôle émis par l'étage decode. Il est utilisé pour calculer la signature associée à chaque instruction dans le module CCFI. Comme indiqué dans la section 3.3, pour GPSA, chaque instruction est associée à une unique signature. La signature étant calculée à partir du pipeline state, chaque instruction doit donc être associée à une unique valeur de pipeline state. Nous appelons cette propriété l'*unicité du pipeline state*.

Dans l'objectif d'assurer l'intégrité du code, le pipeline state doit représenter l'ensemble des bits qui composent l'instruction dont il résulte. De plus, GPSA se base sur la comparaison, lors de l'exécution, de signatures à des signatures de références préalablement calculées. Le pipeline state doit donc aussi être calculable de manière déterministe à partir du code du programme par analyse statique. Cela signifie que le pipeline state est composé de signaux de contrôle déterministes, indépendants des données et issus du décodage des instructions.

On peut différencier deux types de signaux de contrôle :

— Les signaux de contrôle statiques ;

— Les signaux de contrôle dynamiques.

Les signaux de contrôle statiques dépendent uniquement de l'instruction dans l'étage decode. Ces signaux peuvent être directement intégrés dans le pipeline state, car leur valeur est uniquement liée à la connaissance de l'instruction dans l'étage decode. On trouve notamment les signaux de contrôle suivant :

— Les signaux de lecture et de sélection des opérandes ;

— Les signaux d'écriture et de sélection du registre de destination ;

— Les signaux de contrôle des étages suivants (ALU, unité mémoire).

Pour compléter l'intégrité du code, il est nécessaire d'ajouter, dans le pipeline state, les signaux représentant les valeurs immédiates, en particulier le décalage d'adresse (*offset* en anglais) utilisé lors des chargements mémoires.

Les signaux de contrôle dynamiques dépendent des données manipulées et des autres instructions dans le pipeline, leur valeur n'est donc pas déterminable statiquement. Les signaux dépendant des données, tels que la décision d'un branchement, ne peuvent pas être intégrés dans le pipeline state. Les signaux de contrôle qui dépendent d'autres instructions présentes dans le pipeline peuvent être intégrés au pipeline state sous certaines conditions. Pour le type de processeurs ciblés dans nos travaux, ces signaux sont limités aux signaux du mécanisme de *forwarding*. Le mécanisme de forwarding permet de ne pas attendre l'écriture du résultat d'une

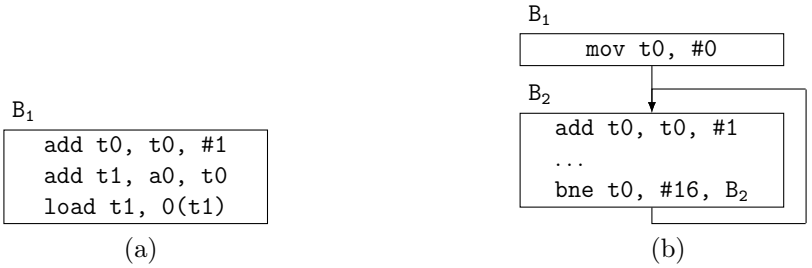

FIGURE 3.4 – Illustration du forwarding : intra (gauche) et inter (droite) blocs de base

instruction dans le banc de registres par l'étage write-back lorsqu'une nouvelle instruction dépend de ce résultat. Le mécanisme de forwarding dépend de l'implémentation du processeur, mais sans perte de généralité, nous faisons l'hypothèse que ces signaux sont calculés dans l'étage decode. Avec la seule connaissance des séquences d'instructions, il est alors possible de précalculer ces signaux et donc de les inclure dans le pipeline state. Notons que, si les signaux de forwarding sont calculés plus tard dans le pipeline, il est possible de les protéger en utilisant le module CSI. La figure 3.4 illustre deux cas pour lesquels le forwarding est utilisé. Dans la figure 3.4a, le forwarding est activé entre les deux instructions `add` du même bloc de base. Cette séquence d'instructions est invariable pour un programme et peut être calculée statiquement. Dans la figure 3.4b, le forwarding est activé lors de la transition $\texttt{B}_1 \rightarrow \texttt{B}_2$ entre les instructions `mov` et `add` mais désactivé lors de la transition $\texttt{B}_2 \rightarrow \texttt{B}_2$ entre les instructions `bneq` et `add`. Ce cas illustre que le forwarding peut être activé entre deux blocs de base. Il n'est alors plus possible de calculer les signaux de contrôle du forwarding statiquement pour l'instruction `add t0, t0, #1` dans $\texttt{B}_2$. Dans ce cas, la dépendance de données entre les instructions de deux blocs de base doit être supprimée pour assurer l'unicité des pipeline states, par exemple par l'addition de nouvelles instructions comme présenté dans la section 4.4.1.

3.4.3 Module CCFI : intégrité du code et du flot de contrôle

Le module CCFI implémente le support matériel de GPSA. Pour cela, deux fonctions sont nécessaires, l'une pour le calcul de la signature et l'autre pour la mise à jour des signatures aux jonctions de chemins de contrôle. Ces deux fonctions doivent posséder certaines propriétés déjà décrites par WERNER et al., [WERNER et al. 2015].

Fonction de signature

Comme présenté sur la figure 3.3, le module CCFI calcule une signature S_i lors de l'exécution à partir du pipeline state courant Σ_i et de la signature précédente S_{i-1}. La signature est conservée dans un registre, appelé *registre de signature*, dans le module CCFI. La valeur de ce registre ne doit pas être directement accessible ni en lecture, ni en écriture pour limiter la surface d'attaque de SCI-FI. La capacité de détection des fautes de CCFI dépend directement des propriétés de f :

— Résistance aux collisions : un attaquant ne doit pas pouvoir forger un bloc de base fauté tel que sa signature soit la même que celle attendue pour le bloc de base original. Ici, on

parle de résistance à la seconde préimage, c'est-à-dire que pour un couple (bloc de base, signature) donné, il n'est pas possible de trouver un second bloc de base qui donne la même signature. Cette propriété empêche également un attaquant de revenir à un état valide de la signature après l'introduction d'une ou plusieurs fautes ;
— Préservation de l'erreur : l'introduction d'une faute dans la signature n'est pas annulée par le traitement d'une séquence d'instructions sans faute. Cette propriété, en combinaison avec la résistance aux collisions, permet le placement arbitraire de vérification de signature dans un programme ;
— Non-associativité : deux séquences contenant les mêmes instructions dans un ordre différent produisent deux signatures différentes. Cette propriété permet d'assurer l'intégrité du flot de contrôle à grain fin, c'est-à-dire que si deux instructions sont permutées, la faute sera détectée ;
— Inversibilité : l'inversibilité de la fonction f est nécessaire afin de pouvoir placer arbitrairement les mises à jour de la signature dans le programme. Pour SCI-FI, cette propriété n'est pas nécessaire, car les mises à jour sont toujours effectuées en fin de bloc de base plutôt que sur une signature intermédiaire.

Il existe plusieurs fonctions possédant ces propriétés, ce qui permet donc de faire des compromis lors de l'implémentation. Les fonctions de hachage cryptographiques remplissent l'ensemble des propriétés requises par SCI-FI. Par exemple, les fonctions de hachage à clé telles que les *Message Authentication Codes* (MAC) sont de bons candidats pour f. Les MACs utilisent une clé secrète, ce qui empêche donc un attaquant de calculer la signature sans la connaissance de la clé. De plus, les MACs permettent au module CCFI d'assurer l'authenticité du code, c'est-à-dire que le code a été émis par une source autorisée (qui connait la clé secrète). Certaines fonctions f offrent une résistance aux collisions plus faible que les fonctions cryptographiques, mais sont tout de même intéressantes dans le contexte de SCI-FI ou des attaques par injection de fautes. C'est le cas des codes détecteurs d'erreurs qui permettent la détection d'un nombre minimum d'inversions de bit. Ces fonctions sont inversibles, ce qui signifie qu'un attaquant peut identifier une faute qui crée une collision. Cependant, le nombre d'inversions de bit nécessaire pour créer une collision sera borné par la capacité de détection de la fonction. La complexité d'une attaque peut donc être fixée par le nombre d'inversions de bit nécessaire pour créer une collision plutôt que par la complexité d'identification de la collision.

Fonction de mise à jour

Cette fonction manipule directement le registre de signature et doit posséder les propriétés suivantes :

— Surjection : pour tout couple (S, IV), il existe au moins un *patch* tel que $IV = u(S, patch)$;
— Préservation de l'erreur : une faute introduite dans la signature ne peut pas être annulée par une mise à jour non fautée ;
— Inversibilité : la fonction u doit être inversible afin de pouvoir calculer le patch à partir de la signature et du vecteur d'initialisation : $patch = u^{-1}(S, IV)$

Mécanisme de mise à jour

Pour appliquer la fonction de mise à jour au registre de signature, il est nécessaire de mettre en place du support pour le logiciel. Il faut d'une part indiquer au module CCFI pour quels blocs de base la mise à jour est appliquée et d'autre part indiquer quel patch utiliser pour la mise à jour. Nous proposons d'étendre le jeu d'instructions du processeur pour ajouter une instruction dédiée à la mise à jour. Cette instruction permet de charger le pacth associé au bloc de base depuis la mémoire dans un registre, appelé *registre de patch*, interne au module CCFI. De plus, la sémantique des instructions de transfert de flot de contrôle est modifiée pour appliquer la mise à jour lorsque le branchement est pris et remettre la valeur par défaut dans le registre de patch. Ainsi, il n'est pas nécessaire d'indiquer au module CCFI pour quels blocs de base la mise à jour doit être appliquée, car elle est appliquée automatiquement lorsqu'un transfert de flot de contrôle est pris. La valeur par défaut du registre de patch doit être connu lors du calcul des signatures de référence. Si la fonction de mise à jour u possède un élément *identité*, c'est-à-dire pour lequel u se comporte comme la fonction identité, alors cette valeur peut être utilisée comme valeur par défaut.

Par conception, notre mécanisme de mise à jour impose l'IV_i d'un bloc de base B_i avec la signature du bloc de base prédécesseur B_f placé immédiatement avant dans l'agencement de la mémoire lorsque la transition entre B_f et B_i est un transfert de flot de contrôle implicite (branchement non pris). Les patchs des autres prédécesseurs sont calculés pour que la signature soit égale à celle de B_f après l'application de la mise à jour. Parfois, le bloc de base placé immédiatement avant B_i n'est pas un prédécesseur de B_i. Dans ce cas, IV_i peut être soit fixé par un autre prédécesseur choisit arbitrairement soit choisit aléatoirement en mettant à jour la signature de tous les prédécesseurs.

Nous proposons de reprendre le CFG exemple de la figure 3.2 en considérant le mécanisme de mise à jour de SCI-FI. Le mécanisme de mise à jour applique u au registre de signature entre les blocs de base $\texttt{B}_1$ et $\texttt{B}_3$, et $\texttt{B}_2$ et $\texttt{B}_4$. Il n'y a pas de patch chargé dans le bloc $\texttt{B}_1$ et donc $IV_{3.0} = u(S_{1.n}, P_{defaut})$. Si la valeur de patch par défaut est l'élément identité de u alors $IV_{3.0} = u(S_{1.n}, P_{defaut}) = S_{1.n}$. Le patch P_2 est chargé dans $\texttt{B}_2$ ce qui implique que le registre de signature est mis à jour avec $u(S_{2.n}, P_2)$ entre $\texttt{B}_2$ et $\texttt{B}_4$. Il n'y a pas de branchement pris entre les blocs de base $\texttt{B}_1$ et $\texttt{B}_2$, et $\texttt{B}_3$ et $\texttt{B}_4$, la fonction de mise à jour n'est donc pas appliquée sur ces chemins.

Mécanisme de vérification

Avec GPSA, chaque instruction exécutée est associée à une signature unique qui dépend de toutes les instructions précédemment exécutées. Grace aux propriétés de préservation de l'erreur des fonctions de signature f et de mise à jour u, toute erreur capturée lors du calcul de la signature sera propagée dans toutes les signatures suivantes. Il est donc possible de placer les vérifications de signature n'importe où dans le programme sans diminuer la capacité de détection des fautes du module CCFI. La stratégie de placement des vérifications n'impacte que le délai de détection de la faute.

SCI-FI utilise des instructions dédiées pour le chargement de la signature de référence depuis la mémoire et le déclenchement de la vérification. Nous les appelons *instructions de vérification* par la suite. L'ajout d'une instruction de vérification implique aussi la déclaration d'une signa-

ture de référence, ce qui impacte donc la taille du code et le temps d'exécution du programme.

Le choix d'utiliser des instructions de vérification permet une grande flexibilité pour la stratégie de vérification d'un programme. Il est donc possible de régler précisément le compromis entre l'augmentation de la taille du code et du temps d'exécution et le délai de vérification. Il est par exemple possible de ne placer qu'une vérification au point de sortie du programme sans réduire la couverture de la protection dans l'objectif de minimiser le surcoût.

3.4.4 Module CSI : intégrité des signaux de contrôle

Nous avons présenté jusqu'ici le module CCFI qui protège les étages avant (fetch, decode et execute) du pipeline. Le module CSI complète la couverture des étages arrière (memory et write-back). Pour cela, le module CSI est placé en parallèle des étages arrière, comme présenté dans la figure 3.2. Il utilise un mécanisme de redondance tel que de la duplication ou des encodages de signaux différents pour protéger la propagation des signaux de contrôle du pipeline state. Un mécanisme permet de détecter les fautes en vérifiant les propriétés du mécanisme de redondance. Pour une implémentation de type duplication, cela représente une comparaison entre les signaux dupliqués et les signaux originaux. Pour une implémentation avec un encodage redondant, cela nécessite de recalculer la valeur des bits redondants, par exemple pour un bit de parité. La vérification du mécanisme de redondance peut être placée soit entre chaque étage, soit dans le dernier étage du pipeline. Ce choix impactera le délai de détection de la faute ainsi que la surface matérielle occupée par le module CSI.

Il est important de noter que le module CSI seul ne permet pas d'assurer l'intégrité des signaux de contrôle du pipeline state. C'est la combinaison des modules CCFI et CSI qui permet dans un premier temps de valider la génération du pipeline state puis la propagation des signaux qui le composent dans la microarchitecture. Le module CSI assurant uniquement l'intégrité de la propagation des signaux de contrôle, il n'est pas nécessaire d'ajouter des métadonnées qui décrivent le comportement de ces signaux. L'avantage majeur de cette solution est la capacité de prendre en compte les évènements dynamiques, comme les défauts de cache ou les mauvaises prédictions de branchement, dans le module CSI. De plus, le module CSI n'impacte que la logique de contrôle dont la surface matérielle est donc faible en comparaison à la surface de la logique qui manipule des données.

3.4.5 Sécurisation des branchements indirects

Le mécanisme décrit jusqu'ici supporte uniquement les branchements directs et de retours de fonction (cas particulier de branchements indirects). Dans cette section, on se concentre sur le support des appels de fonction indirects. Les autres branchements indirects peuvent être supprimés lors de la compilation en indiquant au compilateur de ne pas utiliser de table de saut, une optimisation courante pour les expressions `switch` du langage C.

Comme présenté dans la section 2.2 pour l'intégrité du flot de contrôle, les branchements indirects représentent une grande partie des vulnérabilités. En effet, un branchement indirect dépend des données et peut potentiellement cibler n'importe quelle adresse. Un attaquant

capable d'altérer arbitrairement la donnée qui représente l'adresse cible d'un branchement indirect peu donc altérer arbitrairement le flot de contrôle. Pour réduire la surface d'attaque sur les branchements indirects, il est nécessaire d'imposer des contraintes sur les adresses qu'un branchement indirect peut cibler. Dans le contexte des appels de fonction indirects, il est possible d'utiliser des classes d'équivalence pour regrouper les adresses valides à l'aide de propriétés communes, par exemple le prototype de la fonction. Une fois les cibles des appels de fonction indirects identifiées, il est nécessaire de vérifier la validité de la cible lors de l'exécution.

GPSA permet de contraindre les chemins d'exécution valides à l'aide du chainage des signatures entre blocs de base. Cependant, plusieurs problèmes apparaissent en présence d'appels de fonction indirects. L'ensemble des fonctions faisant parties de la même classe d'équivalence partagent les mêmes prédécesseurs et donc le même IV. Cela signifie qu'une attaque qui appelle une autre fonction de la classe d'équivalence que celle attendue à un moment précis de l'exécution ne sera pas détectée par GPSA. Cette attaque peut être réalisée en altérant le registre qui contient l'adresse de la fonction cible, mais cela sort de notre modèle de menace qui considère uniquement les attaques sur le chemin d'instruction (section 3.2). Il est aussi possible d'attaquer le registre de compteur de programme (PC) pour modifier arbitrairement l'adresse qu'il contient. Cette attaque peut être détectée en attribuant un IV unique à chaque fonction. De plus, les blocs de base suivant les sites d'appel de fonction indirect ont pour prédécesseurs les blocs de base de retour de toutes les fonctions de la classe d'équivalence associée au site d'appel. Comme pour les fonctions faisant partie d'une même classe d'équivalence, les blocs de base qui suivent les appels de fonction indirects partagent le même IV, ce qui ne permet pas de distinguer quelle fonction a été exécutée en vérifiant la signature. Pour réduire la surface d'attaque sur les appels de fonction indirects et les retours qui suivent, il est nécessaire d'associer un IV unique et une signature de sortie unique à chaque fonction.

Dans SCI-FI, les appels de fonction indirects sont remplacés par des séquences de branchements conditionnels et d'appels de fonction directs lors de la compilation. Nous appelons ces séquences des *dispatchers*, et nous présentons un exemple dans le listing 3.2. La fonction `foo` prend en paramètre un pointeur de fonction `fptr` qui pointe vers la classe d'équivalence `EC0` constituée des fonctions `bar` et `baz`. L'appel de fonction indirect dans `foo` est remplacé par un appel direct au dispatcher `EC0_a0` qui réalise un appel direct vers une fonction de la classe `EC0` dont l'adresse de la cible est contenue dans le registre `a0`. Les deux premières instructions `push` du dispatcher permettent de sauvegarder l'adresse de retour et le contenu d'un registre temporaire dans la pile. Le registre temporaire est utilisé pour comparer le contenu du registre `a0` aux adresses des fonctions `bar` et `baz`. Si le registre `a0` contient l'adresse d'une de ces deux fonctions alors le dispatcher réalise un appel direct à la fonction. Sinon, le dispatcher appel le gestionnaire d'erreur `error_handler` pour signaler un problème. Au retour de l'appel, le dispatcher restaure la pile et rend le contrôle à la fonction qui l'a appelée, ici la fonction `foo`.

Listing 3.1 – Exemple d'un appel de fonction indirect dans le langage C

```
void bar();
void baz();

/* fptr pointe soit vers bar,
 *              soit vers baz */
void foo(void (*fptr)()) {
  fptr();
}
```

Listing 3.2 – Protection du code source du listing 3.1 avec SCI-FI (en pseudo assembleur RISC-V)

```
foo:
  call  EC0_a0
  ret

EC0_a0:
  push  ra
  push  s11
EC0_a0_bar:
  li    s11, bar
  bne   s11, a0, EC0_a0_baz
  call  bar
  jmp   EC0_a0_ret
EC0_a0_baz:
  li    s11, baz
  bne   s11, a0, error_handler
  call  baz
EC0_a0_ret:
  pop   ra
  pop   s11
  ret
```

Comme les dispatchers utilisent uniquement des appels de fonction directs chaque fonction peut être associée à un IV unique. Les dispatchers résolvent aussi le problème des IVs des blocs de base qui suivent les sites d'appel de fonction indirect. En effet, chaque site d'appel ne peut appeler qu'une seule fonction grâce à l'élimination des appels de fonction indirects par les dispatchers. La seule vulnérabilité restante sur les branchements indirects serait l'altération du registre qui contient l'adresse de la cible du branchement pour obtenir une autre adresse valide. Pour les retours de fonction, ce problème est souvent résolu par l'utilisation d'une *shadow stack*, c'est-à-dire une duplication de la pile pour les adresses de retour de fonction [Ozdoganoglu et al. 2006].

Pour certaines applications, il n'est pas possible de supprimer les branchements indirects lors de la compilation. C'est par exemple le cas pour les systèmes de démarrage pour lesquels le transfert de flot de contrôle entre le système de démarrage et le programme applicatif est réalisé à l'aide d'un branchement indirect. Le problème est alors que le système de démarrage sécurisé et le programme applicatif sont compilés indépendamment, ce qui empêche l'élimination du branchement indirect entre les deux lors de la compilation. De plus, le programme applicatif est souvent amené à être mis à jour une fois le système déployé, ce qui peut modifier son adresse dans la mémoire.

Une solution à ce problème est d'utiliser une mise à jour dont l'adresse du patch est relative à la cible du branchement indirect. Par exemple, le patch pourrait être placé en entête du bloc de base cible dans la mémoire de code. Il est tout de même nécessaire de définir une signature

d'interface pour le branchement indirect afin de calculer le patch pour la destination. Il est donc nécessaire de connaitre la signature du bloc de base contenant le branchement indirect lors de la compilation du code de destination. Il faut également noter qu'un attaquant capable de générer des séquences d'instructions avec des signatures valides, par exemple avec une fonction de signature non cryptographique, pourrait dérouter le flot de contrôle vers une nouvelle séquence d'instructions sans que cela ne soit détecté.

3.4.6 Prédiction de branchement

Bien que peu courant pour le type de processeur visé, il est possible que le pipeline d'un processeur implémente un mécanisme de prédiction de branchement. Ce mécanisme permet d'optimiser le temps d'attente lors des branchements conditionnels en commençant par exécuter le chemin calculé comme le plus probable. Dans le cas d'une mauvaise prédiction, les instructions dans le pipeline sont invalidées et l'exécution reprend à l'adresse qui correspond à la cible correcte du branchement.

Pour être compatible avec la prédiction de branchement, le module CCFI doit sauvegarder le registre de signature après l'exécution d'un branchement. La signature est sauvegardée avec le résultat de la fonction de mise à jour si le branchement est prédit "non pris" de telle sorte que la signature soit restaurée dans l'état "branchement pris" en cas de mauvaise prédiction. À l'inverse, si le branchement est prédit "pris", alors la signature est sauvegardée à partir du registre de signature.

À la suite d'une mauvaise prédiction, les instructions dans le processeur sont invalidées. Cette opération peut impacter les signaux de contrôle dynamiques des instructions exécutées ensuite. Cela peut donc avoir un effet sur la propriété d'unicité du pipeline state décrite dans la section 3.4.2. Cependant, SCI-FI impose que les dépendances entre instructions soient éliminées lors des transitions entre blocs de base pour maintenir la propriété d'unicité du pipeline state lorsque plusieurs chemins se rejoignent. Il est donc nécessaire de supprimer les éventuelles dépendances entre instructions restantes aux frontières des blocs de base. Pour le type de processeur visé par SCI-FI cela ne représente pas un surcoût important. Il est probable que le surcoût pour éliminer les dépendances aux frontières des blocs de base augmente fortement pour des processeurs avec un pipeline plus profond.

Il est également nécessaire de protéger les instructions invalidées après une mauvaise prédiction de branchement, par exemple contre une faute qui réactiverait ces instructions. Il est donc nécessaire d'intégrer les signaux de prédiction de branchement dans le module CSI pour assurer la propagation des signaux même une fois invalidés.

En conclusion, SCI-FI peut s'adapter à un processeur qui implémente la prédiction de branchement au prix d'un léger surcoût.

Il faut également noter que le mécanisme de prédiction de branchement lui-même est sensible aux attaques par injection de faute. SCI-FI n'est pas capable de détecter une faute sur le signal de mauvaise prédiction. Il est donc nécessaire de protéger ce signal par exemple avec un mécanisme de redondance semblable au module CSI. Toute autre faute sur le mécanisme de prédiction de branchement affectera le flot de contrôle et sera donc détectée par SCI-FI.

3.4.7 Sécurisation du mécanisme d'interruptions

Les interruptions sont des évènements qui peuvent survenir à n'importe quel moment lors de l'exécution d'un programme. Les gestionnaires d'interruptions ne peuvent donc pas être associés à des instructions prédécesseures. SCI-FI nécessite un mécanisme dédié au support des interruptions, ainsi qu'à leur sécurisation.

Dans SCI-FI, chaque gestionnaire d'interruptions est associé à un IV différent et unique. Tous les IVs sont stockés en mémoire dans un tableau de façon similaire au tableau de vecteur d'interruptions. Lors du déclenchement d'une interruption, le registre de signature est sauvegardé dans un registre dédié que nous appelons le *registre de contexte*. Le module CCFI sélectionne ensuite l'IV qui correspond à l'interruption déclenchée pour réinitialiser le registre de signature. Enfin, le processeur peut commencer l'exécution du gestionnaire d'interruptions. Une instruction de vérification peut être placée à la fin du gestionnaire d'interruption pour vérifier son intégrité. Il est aussi possible de placer des instructions de vérification dans le code du gestionnaire d'interruptions comme pour les séquences d'instructions classiques. Cela permet notamment de réduire le délai entre deux vérifications et donc le délai de détection des fautes. Lorsque le gestionnaire d'interruptions se termine, le registre de signature est restauré avec la valeur contenue dans le registre de contexte.

Après le retour d'un gestionnaire d'interruptions, les dernières instructions de celui-ci sont encore dans le pipeline. Cela peut impacter l'unicité du pipeline state de la même façon que les dépendances entres blocs de base comme présenté en section 3.4.2. Pour éviter ce problème, SCI-FI retarde le traitement des interruptions aux frontières de bloc de base où les dépendances entre bloc de base sont toujours supprimées.

Dans SCI-FI, le registre de signature n'est jamais sauvegardé en mémoire. Cela permet d'éviter une partie des attaques sur la signature sauvegardée au détriment d'un léger surcoût. Pour le support des interruptions imbriquées, il est possible de remplacer le registre de contexte par une pile de contexte.

3.5 Support pour des microarchitectures plus complexes

Nous avons jusqu'à présent fait l'hypothèse que le processeur à sécuriser est composé d'un pipeline limité à quelques étages, traitant une seule instruction à la fois avec de l'exécution dans l'ordre. Dans cette section, nous proposons de discuter des évolutions nécessaires pour supporter des microarchitectures plus complexes.

De nombreuses unités logiques dont le calcul est réparti sur plusieurs cycles sont pipelinées pour améliorer les performances du processeur. Dans ce cas, la profondeur du pipeline augmente en fonction des unités logiques qui le composent. Une augmentation de la profondeur du pipeline implique une augmentation de la complexité du mécanisme de forwarding. Si le nombre d'étages du pipeline augmente, le nombre d'instructions dont une instruction peut dépendre augmente proportionnellement. Pour assurer l'unicité du pipeline state entre les blocs de base, il est nécessaire de supprimer les dépendances entres instructions aux frontières des blocs de base. L'augmentation du nombre d'étages dans un pipeline augmente donc le coût de l'élimination des dépendances entre instructions aux frontières des blocs de base.

Pour réduire le temps d'exécution d'un processeur, une solution est d'exécuter plusieurs ins-

tructions par cycle (*multiple issue*). Il existe deux méthodes pour exécuter plusieurs instructions par cycle :

— Statique : les processeurs *Very Long Instruction Word* (VLIW) pour lesquels le compilateur est responsable de l'ordonnancement des instructions ;
— Dynamique : les processeurs superscalaires pour lesquels le pipeline peut ordonnancer les instructions ;

Pour les processeurs VLIW, il est nécessaire d'augmenter la taille du pipeline state pour prendre en compte tous les signaux de contrôle. Les signaux de contrôle pour le forwarding peuvent toujours être pris en compte puisqu'ils peuvent être déterminés statiquement lors de la compilation. L'impact de l'élimination des dépendances entre instructions aux frontières des blocs de base dépend alors de la profondeur du pipeline.

Pour les processeurs superscalaires il est possible que les instructions soit exécutées dans l'ordre ou dans le désordre. Pour une exécution dans l'ordre et dynamique, il est toujours possible de calculer les signatures en prenant en compte les signaux de forwarding tant que les signaux de forwarding sont indépendants des données manipulées. Enfin, pour l'exécution dynamique dans le désordre, le processeur conserve un fonctionnement dans l'ordre pour le chargement et le décodage des instructions. Ce n'est qu'après le décodage que les instructions sont réordonnancées et distribuées aux unités logiques de calcul. Il est donc possible de calculer la signature dans l'ordre après le décodage des instructions. En ce qui concerne le mécanisme de forwarding, celui-ci est remplacé par le réordonnancement des instructions après l'étage decode. Bien que complètement déterministe, le mécanisme de réordonnancement peut prendre en compte les évènements dynamiques tels que les défauts de cache. Dans ce cas, il n'est pas possible de prédire le comportement de ce mécanisme par une analyse statique du programme. Cela signifie que pour les pipelines dans le désordre, il n'est pas possible d'intégrer les signaux de contrôle du mécanisme de forwarding et du mécanisme de réordonnancement dans le pipeline state. Il est toutefois possible de dupliquer ces mécanismes. Cependant, il sera alors difficile d'obtenir le même niveau de sécurité, par exemple sur le nombre d'inversions de bit nécessaires pour réaliser une faute non détectée.

De nombreuses optimisations de la microarchitecture peuvent augmenter la complexité du pipeline. Il est évident que cela impacte notre contre-mesure. Néanmoins, SCI-FI semble pouvoir s'adapter à des microarchitectures plus complexes. Cette perspective montre le potentiel d'application de SCI-FI, même en dehors du type de processeur originalement visé par nos travaux.

3.6 Discussion et positionnement

SCI-FI se place parmi les contre-mesures qui protègent l'intégrité du code et du flot de contrôle contre les attaques par injection de faute. Les travaux de Werner et al. sont très proches de notre proposition [Werner et al. 2015]. En effet, nous avons réutilisé la technique GPSA qu'ils ont introduite dans le contexte des attaques par injection de fautes. Nous avons aussi repris les propriétés nécessaires pour la fonction de signature et la fonction de mise à jour. Nos travaux se distinguent par la mise en œuvre de GPSA. Nous avons fait le choix d'introduire le pipeline state pour calculer la signature là où [Werner et al. 2015] utilisent

l'encodage des instructions. De plus, nous sommes les premiers à notre connaissance à combiner un mécanisme d'intégrité du code et du flot de contrôle à un mécanisme de protection de la microarchitecture. Cette solution nous permet de protéger efficacement les signaux de contrôle dans la microarchitecture et ainsi de couvrir l'intégralité du chemin d'instructions.

SCI-FI se distingue également par un support complet du matériel au logiciel. En effet, dans nos travaux, nous prenons en compte le support des branchements indirects et des interruptions. De plus, nous discutons de l'impact de SCI-FI sur différentes architectures. Nous proposons ainsi une contre-mesure compatible avec une utilisation pour la protection des systèmes embarqués.

L'étude des attaques par injection de fautes a débuté dans les années 1990. Pourtant, il existait auparavant des problématiques similaires pour la fiabilité, notamment dans le domaine de spatial. Les rayons cosmiques ont été identifiés comme la cause de faute sur les bits dans les systèmes spatiaux tels que les satellites [Binder et al. 1975]. Pour faire face à ce problème de nombreuses solutions ont été proposées pour renforcer la fiabilité des systèmes numériques. En comparaison aux attaques par injection de faute, le modèle de faute est ici probabiliste sans aucun contrôle particulier. Cela signifie que dans le contexte de la fiabilité, une faute ciblera les points les plus vulnérables avec la même probabilité que le reste du système. Dans le contexte des attaques par injection de fautes, cette hypothèse n'est pas valable, car un attaquant ciblera toujours les éléments les plus vulnérables. Il est donc nécessaire d'éliminer ces points faibles résiduels lorsque les fautes peuvent être injectées par un attaquant.

Les travaux de S. Kim et al. proposent de calculer une signature à partir des signaux de contrôle [S. Kim et al. 2001]. Cette solution vise la protection de la logique de contrôle dans un processeur face à des fautes probabilistes. Ils utilisent une fonction non sécurisée (OU exclusif) pour dériver une signature à partir des signaux de contrôle statiques du processeur et un mécanisme de redondance pour protéger les signaux de contrôle dynamique. Lors de la première exécution d'une instruction, la signature générée est placée dans un cache. Pour les exécutions suivantes de cette instruction, la signature calculée est comparée au contenu du cache. Cette solution ne nécessite donc aucune instrumentation du programme avec des signatures de référence. Le problème avec ce type de mécanisme est qu'il n'est pas possible de protéger l'exécution d'une instruction pour laquelle la signature associée n'est pas disponible dans le cache. De plus, cette solution ne permet pas de détecter les fautes qui modifie l'intégrité du code en mémoire ou l'intégrité du flot de contrôle.

Les solutions de fiabilité face aux fautes dans les circuits intégrés ne sont pas conçues pour résister à un attaquant. Il est donc nécessaire de proposer de nouvelles solutions pour répondre aux attaques par injection de faute. SCI-FI s'inspire de ce qui existe déjà dans le domaine de la fiabilité pour construire une contre-mesure résistant à un attaquant qui aurait un contrôle complet sur la faute injectée dans le circuit.

3.7 Conclusion

Dans ce chapitre, nous avons proposé une extension de la microarchitecture et du support logiciel pour protéger un système contre les attaques par injection de faute. Cette contre-mesure assure les propriétés d'intégrité du code, du flot de contrôle et d'exécution. SCI-FI articule deux mécanismes de sécurité pour protéger la logique de contrôle dédiée à l'exécution des instructions

contre les fautes qui ciblent la microarchitecture. Le premier module implémente la technique GPSA en calculant une signature à partir du pipeline state, un ensemble de signaux de contrôle indépendants des données. Le pipeline state permet de représenter fidèlement l'encodage binaire des instructions pour assurer l'intégrité du code. Le chainage des signatures est utilisé pour détecter toute déviation dans le flot de contrôle. Ce module assure donc l'intégrité du code, du flot de contrôle et des signaux de contrôle de l'étage fetch à l'étage decode. Le second module implémente un mécanisme de redondance pour assurer l'intégrité de la propagation des signaux de contrôle composant le pipeline state dans les étages restants du pipeline. Cela permet une couverture complète de l'intégrité des signaux du pipeline state dans la microarchitecture du processeur.

Dans le chapitre suivant, nous proposons deux implémentations de SCI-FI. Nous utilisons ces implémentations pour évaluer le niveau de sécurité apporté par SCI-FI ainsi que l'impact sur les performances matérielles et logicielles.

Chapitre 4

Implémentations et évaluations

Sommaire du présent chapitre

4.1 Introduction

Dans le chapitre 3 nous avons présenté SCI-FI, une nouvelle contre-mesure mixte matérielle et logicielle contre les attaques par injection de fautes. SCI-FI est composée d'une extension de la microarchitecture divisée en deux modules CCFI et CSI. SCI-FI est également accompagnée d'un support logiciel via l'extension du jeu d'instructions du processeur cible et une modification de la chaîne de compilation.

Dans ce chapitre, nous proposons deux implémentations de SCI-FI avec deux fonctions de signature différentes : une cryptographique et une non cryptographique. Nous évaluons ensuite le niveau de sécurité et les performances matérielles et logicielles de ces deux implémentations.

La section 4.2 présente le processeur cible que nous choisissons pour nos implémentations de SCI-FI. Dans la section 4.3, nous décrivons les modifications apportées pour supporter SCI-FI sur ce processeur. Ensuite, la section 4.4 présente la chaine de compilation et le support logiciel que nous avons mis en œuvre. La section 4.5 rapporte notre analyse du niveau de sécurité assurée par nos implémentations. Enfin, dans la section 4.6, nous rapportons les résultats de nos évaluations des performances de SCI-FI aussi bien au niveau logiciel que matériel.

4.2 Processeur cible : CV32E40P

Pour nos implémentations de SCI-FI, nous choisissons le processeur CV32E40P comme base. C'est un processeur qui implémente le jeu d'instructions RISC-V version 2.1 et plus particulièrement le jeu RV32IMC, c'est-à-dire un jeu d'instructions 32 bits comprenant les extensions pour les instructions compressées sur 16 bits et le support pour la multiplication entière [Waterman et al. 2019]. Il est composé d'un pipeline de 4 étages comprenant les étages fetch (IF), decode (ID), execute (EX) et write-back (WB), présenté en figure 4.1. L'étage fetch est composé d'un *prefetch buffer* qui charge les instructions depuis la mémoire, d'un module *aligner* pour réaligner les mots lorsque les instructions 32 bits et 16 bits sont mélangées et d'un décodeur pour transformer les instructions compressées en instructions 32 bits. L'étage decode comprend la logique pour contrôler les unités de calcul et sélectionner les opérandes dans le banc de registres. C'est aussi dans l'étage decode que se situe le mécanisme de forwarding. L'étage execute comprend trois unités : l'unité arithmétique et logique (ALU), l'unité de transaction mémoire (LSU) et l'unité de multiplication entière (MULT). Les unités de transaction mémoire et de multiplication peuvent toutes les deux prendre plusieurs cycles pour exécuter une instruction. Enfin, l'étage write-back écrit le résultat de l'instruction dans le banc de registres. Il est possible d'écrire le résultat de deux instructions en même temps si l'une utilise l'unité arithmétique et logique ou le multiplicateur et l'autre utilise l'unité de transaction mémoire. L'unité CSR en jaune qui est située dans l'étage execute est un banc de registres de status qui permet de contrôler le comportement du processeur.

Nous avons choisi le CV32E40P car il correspond au type de processeurs visés par SCI-FI comme présenté dans le chapitre 3, section 3.2. De plus, le jeu d'instructions RISC-V est conçu pour être facilement étendu, ce qui est nécessaire pour ajouter les instructions de chargement de patch et de vérification dédiées à SCI-FI. Enfin, le CV32E40P est un processeur régulièrement utilisé dans les travaux de recherche, par exemple [Werner et al. 2018 ; Savry et al. 2020], et réutiliser ce processeur permet de comparer plus facilement les résultats que nous obtenons

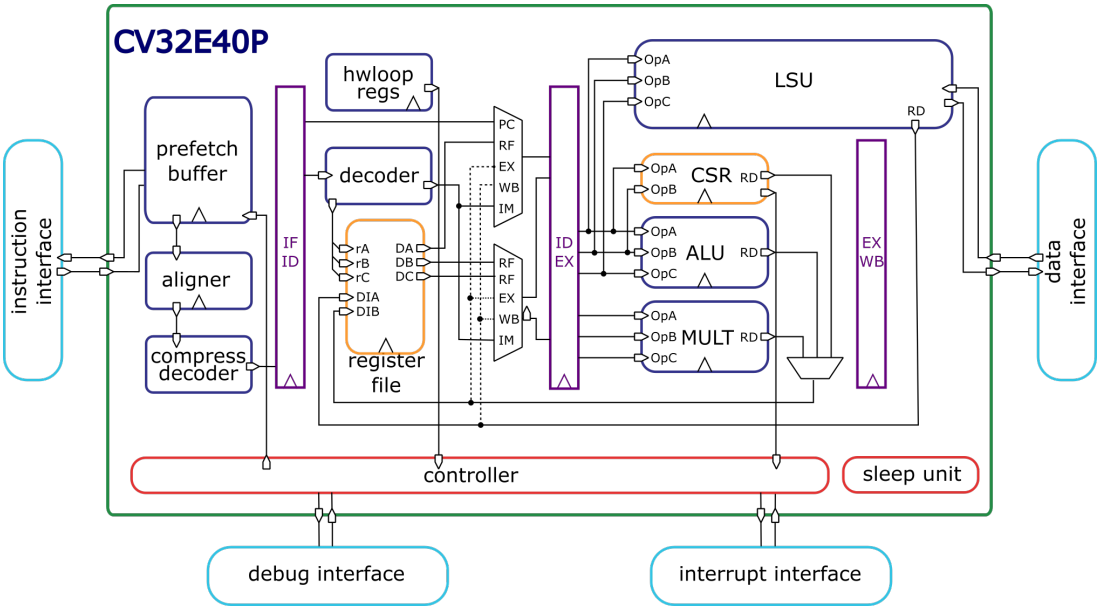

FIGURE 4.1 – Organisation du pipeline du processeur CV32E40P

avec ces autres travaux.

Nous décidons de ne pas utiliser le support des instructions compressées lors de l'implémentation de SCI-FI. Les instructions compressées sont transformées en instructions 32 bits dans l'étage fetch avant d'être décodées dans l'étage decode. Leur impact sur SCI-FI est donc limité puisque les instructions compressées ne modifieraient que la taille du code. Nous n'utilisons pas non plus les instructions de multiplication entière. L'unité de multiplication (MULT) étant plus complexe que l'unité arithmétique et logique les calculs peuvent s'étendre sur plusieurs cycles. Nous faisons le choix de nous concentrer sur des unités de calcul plus simples dans un premier temps pour valider le concept du module CSI.

4.3 Extensions du CV32E40P avec SCI-FI

SCI-FI s'intègre directement dans le pipeline du processeur comme présenté dans la figure 4.2. Nous ajoutons le module CCFI en parallèle de l'étage execute et le module CSI en parallèle des étages execute et write-back. L'étage execute est à la fois couvert par le module CCFI et CSI, car une partie des signaux qu'il reçoit sont directement émis par l'étage decode, et l'autre partie est utilisée sur plusieurs cycles pour réaliser les transactions mémoire. Nous décrivons dans cette section les signaux de contrôle retenus pour la création du pipeline state, les fonctions utilisées pour les modules CCFI et CSI ainsi que les modifications apportées au pipeline du CV32E40P.

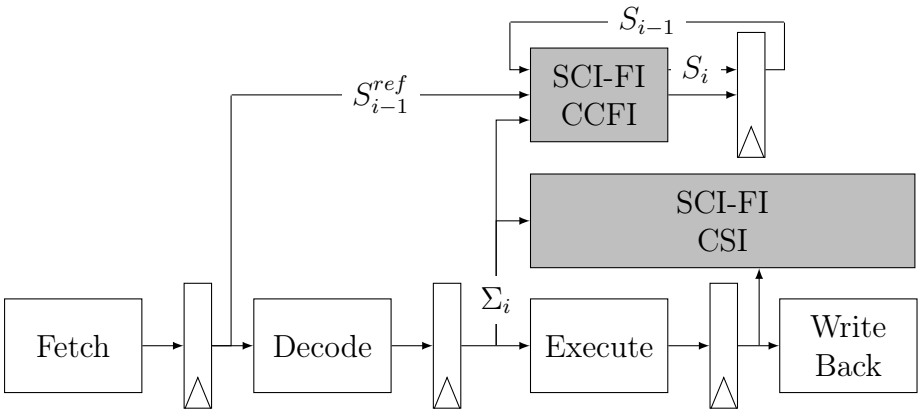

FIGURE 4.2 – Extensions du CV32E40P avec SCI-FI (modules gris)

4.3.1 Pipeline state

Le pipeline state est un vecteur de bits composé de signaux de contrôle émis par l'étage decode. Il est utilisé dans le module CCFI pour calculer la signature. Lors de la construction du pipeline state les objectifs sont d'inclure tous les signaux de contrôle non redondants qui pilotent les unités de calcul. Il est aussi important que le pipeline state couvre tout l'encodage de l'instruction binaire pour assurer la propriété d'intégrité du code. Cependant, on souhaite limiter la taille du pipeline state pour limiter la taille du circuit nécessaire au calcul de la signature. Cela implique qu'il est nécessaire de faire une analyse complexe pour sélectionner les signaux de contrôle du pipeline state. Il n'existe pas encore d'outil permettant de construire le pipeline state automatiquement. Nous faisons donc une analyse manuelle des signaux pour choisir ceux que nous utilisons pour la construction du pipeline state.

Pour nos implémentations de SCI-FI dans le CV32E40P, le pipeline state est constitué de 64 bits de signaux de contrôle :

1. Tous les signaux de contrôle non redondants, internes à l'étage decode, qui sélectionne les opérandes de l'instruction :
   — 23 bits contrôlant les multiplexeurs de sélection des opérandes ;
   — 4 bits contrôlant les multiplexeurs du mécanisme de forwarding ;
2. Tous les signaux de contrôle produits par l'étage décode et transmis aux unités logiques dans les étages suivants :
   — 7 bits contrôlant l'ALU ;
   — 2 bits contrôlant la LSU ;
   — 10 bits contrôlant l'écriture dans le banc de registres ;
3. Tous les signaux dérivés des valeurs immédiates encodées dans l'instruction, ce qui représente 10 bits. Bien que le jeu d'instructions RV32I supporte des valeurs immédiates dont la taille peut aller jusqu'à 20 bits, les 10 bits restants sont déjà couverts part d'autres signaux dans le pipeline state. En effet, suivant le type de l'instruction certains bits peuvent servir soit à coder la sélections des registres des opérandes, soit à coder les valeurs immédiates. Les signaux de contrôle de sélection des opérandes permettent donc de couvrir les valeurs immédiates suivant le type de l'instruction ;

4. 8 bits de bourrage pour compléter les 64 bits du pipeline state.

Bien que nous ne supportons pas les instructions compressées, il ne serait pas nécessaire d'ajouter des signaux de contrôle au pipeline state afin de les protéger. En effet, les instructions compressées sont uniquement traduites en instructions 32 bits avant d'être décodées. Une faute sur une instruction compressée serait donc automatiquement capturée dans le pipeline state après la traduction. Nous n'intégrons pas les signaux de contrôle du multiplicateur, car les instructions de multiplication ne sont pas utilisées pour notre implémentation (section 4.2). Il serait nécessaire d'ajouter quatre bits au pipeline state pour supporter les instructions de multiplication. Comme l'ALU et le multiplicateur ne peuvent pas être utilisés en même temps, il est possible de multiplexer les signaux de contrôle de l'ALU et du multiplicateur dans le pipeline state. Il serait également possible de supporter les instructions associées au CSR de la même façon en ajoutant les trois bits de signaux de contrôle associés. Toutefois, pour ces travaux, nous nous limitons au jeu d'instructions non privilégiées RISC-V, c'est-à-dire RV32I et nous laissons le support des instructions privilégiées pour de futurs travaux. Il est important de noter qu'une instruction ne peut accéder qu'à une seule unité de calcul, il est donc possible de multiplexer les signaux de contrôle dans le pipeline state en fonction de l'unité de calcul activée. Néanmoins, il faut pour cela pouvoir différencier quelle unité de calcul est utilisée dans le pipeline state pour éviter des collisions, par exemple en intégrant les signaux d'activation de ces unités. Ainsi, le support des instructions non considérées dans notre implémentation n'augmenterait pas la taille du pipeline state. De plus, si nécessaire, notre implémentation dispose de huit bits de bourrage pour intégrer de nouveaux signaux de contrôle.

4.3.2 Module CCFI : Fonctions de signature et de mise à jour

Le module CCFI implémente le support matériel pour GPSA. Il est donc composé d'une fonction de signature et d'une fonction de mise à jour pour réaliser les ajustements de signature. Ces fonctions impactent fortement les performances de SCI-FI. D'une part, les propriétés de ces fonctions permettent d'atteindre différents niveaux de sécurité, comme présenté dans le chapitre 3, section 3.4.3. D'autre part, la fonction de signature calcule la signature associée à l'instruction courante en parallèle de l'exécution des instructions et de préférence ne doit pas ralentir le fonctionnement du pipeline. La fonction de signature doit donc s'exécuter en un cycle processeur au maximum et ne pas rallonger le chemin critique du processeur.

Nous proposons deux implémentations de SCI-FI. La première utilise une fonction de signature cryptographique CBC-MAC basée sur l'algorithme de chiffrement par bloc Prince [Borghoff et al. 2012] et permet d'obtenir le niveau de sécurité maximal de SCI-FI. La seconde repose sur l'utilisation d'une fonction de détection d'erreur de type CRC32. Pour les deux implémentations, nous utilisons le OU exclusif comme fonction de mise à jour. Nous choisissons 0 comme valeur par défaut pour le registre de patch, car cela permet de transformer le OU exclusif en fonction identité lorsque aucun patch n'est chargé : $x \oplus 0 = x$.

L'utilisation d'une fonction de signature cryptographique permet d'atteindre le plus haut niveau de sécurité possible avec SCI-FI. Il est en plus possible d'assurer la propriété d'authenticité du code. Nous sélectionnons une fonction cryptographique de type *Message Authentication Code* (MAC). Ce type de fonction permet de calculer une signature, aussi appelée *tag* ou MAC

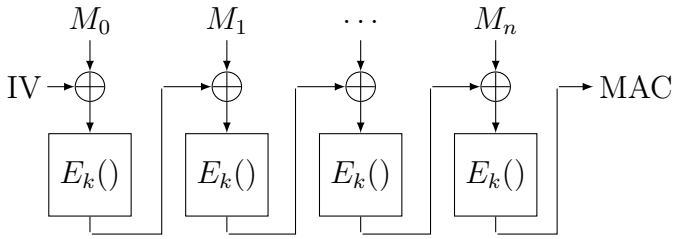

FIGURE 4.3 – Principe de calcul d'un CBC-MAC

dans la littérature, à partir d'un message de taille variable et d'une clé secrète. Lorsqu'un message est envoyé au destinataire avec un MAC, le destinataire peut recalculer le MAC à partir du message reçu et comparer le résultat obtenu au MAC transmis avec le message. Si les deux MACs correspondent alors le destinataire est sûr que le message n'a pas été modifié et qu'il a été émis par une entité qui possède la même clé secrète que lui, ce qui correspond à vérifier l'authenticité du message. Il existe plusieurs types de MAC. Les H-MACs sont construits à partir de fonctions de hachage cryptographiques. Cependant, les fonctions de hachage cryptographiques ne sont pas adaptées à notre utilisation, car il est nécessaire d'utiliser deux fois la fonction de hachage pour calculer le MAC. Les CBC-MACs sont des constructions qui utilisent les algorithmes de chiffrement par bloc et le mode *Cipher Block Chaining* (CBC). L'idée est d'absorber le message bloc par bloc en recombinant la sortie de l'algorithme de chiffrement au bloc de message suivant avec l'opérateur OU Exclusif. Le principe du CBC-MACs est présenté sur la figure 4.3. Dans le cadre de SCI-FI, l'IV du CBC-MAC est la signature du bloc de base précédent et chaque message correspond à une instruction dans le bloc de base.

Nous choisissons de construire un CBC-MAC avec Prince, car c'est un algorithme de chiffrement par bloc léger spécialement conçu pour les implémentations matérielles [BORGHOFF et al. 2012]. Cela nous permet notamment d'utiliser une version entièrement déroulée de l'algorithme Prince afin que la fonction de signature s'exécute en un seul cycle tout en conservant une surface de circuit inférieure aux autres algorithmes de chiffrement par bloc de la même catégorie [MAENE et al. 2016]. Prince est un algorithme qui manipule des blocs de 64 bits qui sont donc compatibles avec la taille de notre pipeline state. Cependant, pour limiter l'emprunte mémoire, nous décidions d'utiliser des signatures de référence de 32 bits, correspondant aux 32 bits de poids faibles des 64 bits de la signature. Nous discutons de l'impact de ce choix dans la section 4.5.

Les fonctions cryptographiques coûtent cher en surface matérielle. Les fonctions de détection d'erreur telles que les CRC sont des candidats idéaux pour des implémentations plus légères. Ces fonctions permettent de détecter un nombre minimum de modifications dans un message de taille donnée. Ces fonctions sont toutefois inversibles et il est possible pour un attaquant de trouver une faute capable de créer une collision dans le calcul de la signature. Le niveau de sécurité apporté par ce type de fonctions est donc dépendant du nombre minimum de bits fautés nécessaires pour créer une collision. Pour chaque instruction, le CRC32 du pipeline state est calculé à partir du pipeline state et du CRC32 précédent.

4.3.3 Module CSI

Pour le module CSI, nous dupliquons tous les signaux transmis aux étages execute et write-back qui ne sont pas directement utilisés au cycle suivant leur émission. Nous utilisons une copie simple comme méthode de redondance. Dans l'étage execute des événements dynamiques de type défaut de cache peuvent survenir lors des transactions mémoire. Cela provoque des cycles de gel pendant lesquels l'exécution des instructions est mise en pause et les signaux de contrôle ne sont pas propagés entre les étages. Il est important de prendre ces cycles de gel en compte d'une part pour assurer le fonctionnement correct du module CSI et d'autre part pour détecter d'éventuelles attaques sur le mécanisme de gestion des évènements dynamiques. La logique de gestion des cycles de gel est donc elle-aussi dupliquée pour assurer que les signaux originaux et dupliqués se propagent de la même façon dans le pipeline.

4.3.4 Mécanisme d'alerte

Les modules CCFI et CSI peuvent tous les deux lever une alerte d'intégrité. Lors de la détection d'une faute, quand la signature est différente de la signature de référence ou quand un signal de contrôle ne correspond pas à sa valeur dupliquée dans le module CSI, une exception matérielle est levée. Pour cela, les signaux de détections des modules CCFI et CSI sont connectés au gestionnaire d'exception. En cas d'alerte, le gestionnaire d'exception suspend l'exécution du programme en cours et exécute une routine logicielle dont l'adresse est indiquée dans le registre CSR `mvtec`. La réponse du système dépend alors de la routine logicielle et le concepteur du système est responsable de la stratégie de réponse lorsqu'une faute est détectée. Il est par exemple possible de reprendre l'exécution depuis le début ou alors de supprimer les données sensibles manipulées par le processeur.

4.3.5 Extension du jeu d'instructions RISC-V pour SCI-FI

SCI-FI nécessite des instructions dédiées pour le chargement des patchs et pour les vérifications. Nous étendons donc le jeu d'instructions RV32I avec une instruction pour le chargement de patch et huit instructions pour les vérifications. Pour rappel, toutes les instructions de flot de contrôle déclenchent le mécanisme de mise à jour lorsqu'un branchement est pris. De plus elles remettent la valeur par défaut dans le registre de patch (branchement pris et non pris). Nous modifions donc également le comportement des instructions de transfert de flot de contrôle du jeu RV32I pour qu'elles réalisent ces opérations. La vérification est uniquement déclenchée suite à une instruction de vérification.

Nous utilisons les instructions *custom*, réservées pour des extensions non standards, pour encoder les instructions dédiées à SCI-FI. Nous choisissons d'utiliser des instructions de flot de contrôle pour implémenter les instructions de vérification. Chaque instruction de vérification correspond à une instruction de flot de contrôle du jeu d'instruction RV32I : `scifi.beq`, `scifi.bne`, `scifi.blt`, `scifi.bge`, `scifi.bltu`, `scifi.bgeu`, `scifi.jal`, `scifi.jalr`. Ces instructions se comportent comme les instructions de flot de contrôle standards, et en plus indiquent à l'étage fetch que le mot mémoire suivant est une signature de référence. À la suite de ces instructions, le module CCFI compare la signature de référence au registre de signature.

En cas de divergence, une exception est levée. Lors de la génération du code, les instructions de vérification remplacent les instructions de flot de contrôle classiques. Ainsi, uniquement une signature de référence est ajoutée dans le code, ce qui limite l'augmentation de la taille du code en comparaison avec une instruction qui chargerait uniquement la signature de référence (une instruction supplémentaire et une signature de référence).

Une instruction de chargement de patch est également ajoutée : `scifi.ldp`. Cette instruction charge le patch à partir d'une adresse de base placée dans un nouveau CSR et d'une valeur de décalage encodée comme un immédiat de 12 bits dans l'instruction. La valeur de décalage sur 12 bits permet d'adresser $2^{14} = 16384$ patchs différents en fixant l'alignement des patchs sur quatre octets (les deux bits de poids faible à 0). L'adresse de base de la section mémoire contenant les patchs est placée dans le CSR `pacth_base` lors du démarrage du processeur. Il est important de noter que les patchs ne peuvent pas être encodés directement dans le champ immédiat de l'instruction. En effet, le champ immédiat de l'instruction est intégré au pipeline state et donc dans la signature. Placer les patchs dans le champ immédiat d'une instruction provoquerait alors une dépendance circulaire entre le calcul de la signature et la détermination des patchs.

Dans notre proposition, les instructions de chargement de patch ajoutent deux mots mémoire (l'instruction de chargement et le patch) et les instructions de vérification ajoutent un mot mémoire (la signature de référence). Il est aussi possible d'utiliser des instructions de flot de contrôle pour charger les patchs et des instructions de chargement depuis la mémoire pour charger les signatures de références. Dans ce cas, les instructions de chargement de patch ajouteraient un mot mémoire (le patch) tandis que les instructions de vérification ajouteraient deux mots mémoire (l'instruction de chargement et la signature de référence). De plus, lors de l'exécution, les instructions de chargement de patch n'ajouteraient qu'un seul cycle (pour le chargement du patch) contre deux cycles pour les instructions de vérification (instruction de type `load`). Suivant le nombre d'instructions de chargements de patch et de vérification dans le programme une solution est donc plus intéressante que l'autre.

Pour ajouter les instructions dédiées à SCI-FI, une alternative à l'extension du jeu d'instructions RISC-V est l'utilisation d'instructions *HINT* du jeu RV32I. Les instructions HINT sont des instructions dont l'exécution n'a aucun effet sur les registres du processeur telles que l'instruction `nop`. Dans le jeu RV32I, l'instruction `nop` est par exemple codée comme une addition immédiate dans le registre zéro (`addi zero, zero, 0`). Il est donc possible de placer les signatures de référence et les patchs dans les bits inutilisés des instructions HINT. Cette solution a pour avantage de limiter les modifications nécessaires sur l'étage decode, car la logique de décodage est déjà présente dans le circuit et il suffit donc de rajouter les connexions (donc des fils) entre le registre d'instruction et le module CCFI. Cependant, il n'est pas possible d'encoder plus de 20 bits de donnée avec les instructions HINT. Il faut alors choisir soit de diminuer la taille des signatures et des patchs, soit utiliser plusieurs instructions pour les charger. De plus, comme expliqué précédemment, placer les patchs dans le champ immédiat d'une instruction provoque une dépendance circulaire entre le calcul de la signature et la détermination des patchs. Il est donc nécessaire de repenser le pipeline state pour que cette solution puisse fonctionner. Il serait aussi possible de ne pas inclure les valeurs immédiates dans le pipeline state spécifiquement pour ces instructions.

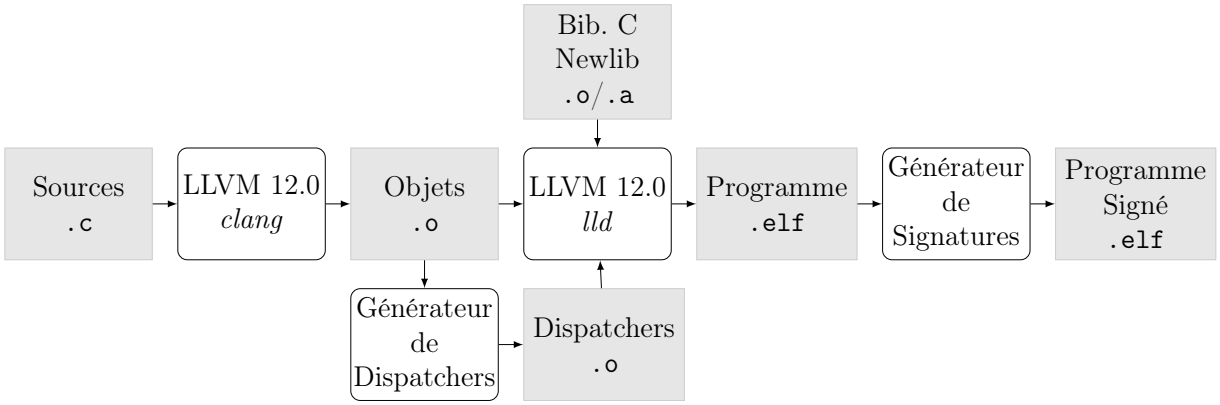

FIGURE 4.4 – Chaîne de compilation de SCI-FI avec les fichiers en gris clair et les outils dans des boîtes arrondies

4.4 Support logiciel

L'extension du processeur CV32E40P avec SCI-FI entraine plusieurs modifications au niveau logiciel. D'une part, nous ajoutons de nouvelles instructions. Il est donc nécessaire d'ajouter le support pour ces instructions dans le compilateur. D'autre part, certaines transformations du code sont nécessaires, notamment pour assurer l'unicité du pipeline state aux frontières des blocs de base ou pour supprimer les appels de fonction indirects.

Une vue d'ensemble du support logiciel de SCI-FI est présenté en figure 4.4. Le code source est transformé en code machine par le compilateur LLVM modifié pour supporter le CV32E40P protégé avec SCI-FI. À partir de ce code machine, le générateur de dispatchers, un outil développé dans le cadre de nos travaux, identifie les appels de fonction indirects, calcule les classes d'équivalences et émet le code correspondant à chaque dispatcher. Nous utilisons ensuite lld, l'outil d'édition des liens de LLVM, pour générer un fichier exécutable ELF à partir du code machine, de la bibliothèque C Newlib et des dispatchers. Enfin, le générateur de signatures, un autre outil développé dans le cadre de nos travaux ajoute les patchs et les signatures de référence au fichier exécutable.

4.4.1 Extension du back-end RISC-V de LLVM

Le back-end de la chaîne de compilation LLVM est étendu avec quatre passes placées juste avant l'émission du code.

Élimination des dépendances entre blocs de base Une première passe élimine les dépendances entre instructions aux frontières des blocs de base. Cette passe est nécessaire pour maintenir la propriété d'unicité du pipeline state présentée en section 3.4.2 et pour supporter les blocs de base avec plusieurs prédécesseurs. En effet, lorsqu'un bloc de base à plusieurs prédécesseurs, le mécanisme de forwarding, dont les signaux de contrôle font partie du pipeline state, peut se comporter différemment en fonction du prédécesseur exécuté. Pour forcer un comportement du mécanisme de forwarding identique pour tous les prédécesseurs, le compilateur élimine les dépendances entre les blocs de base et leurs prédécesseurs. Le processeur CV32E40P possède

quatre étages de pipeline dont deux après l'étage decode. Pour ce processeur, une instruction peut donc dépendre au plus des deux instructions qui la précèdent. De plus, les instructions de flot de contrôle (qui marquent la fin d'un bloc de base) s'exécutent en deux ou trois cycles suivant le type de l'instruction. Il ne peut donc pas y avoir de dépendance entre un bloc de base se terminant par une instruction de flot de contrôle et ses successeurs. Seuls les blocs de base ne se terminant pas par une instruction de flot de contrôle sont donc impactés par cette passe. Notons que le successeur d'un bloc de base ne se terminant pas par une instruction de flot de contrôle a toujours plusieurs prédécesseurs, sinon les deux blocs de base pourraient être fusionnés pour former un seul bloc de base. Cette passe considére donc les deux dernières instructions du prédécesseur et les deux premières instructions du bloc de base successeur. En cas de dépendance détectée, la passe ajoute le nombre minimum d'instructions `nop` nécessaires soit un maximum de deux `nop`.

Élimination des appels de fonction indirects La seconde passe ajoutée au back-end remplace les appels de fonction indirects par des appels directs à des dispatchers. Lors de cette passe, il n'est pas possible d'identifier quelles sont les candidats valides ou la classe d'équivalence associée aux appels de fonction indirects. Chaque appel de fonction indirect est donc associé à son propre dispatcher. La génération des dispatchers est réalisée en dehors du compilateur par le générateur de dispatchers juste avant l'édition des liens.

L'élimination des branchements indirects qui ne sont pas des appels de fonction, par exemple dus aux constructions `switch` dans le langage C, est fait avec l'option `-fno-jump-table` du compilateur.

Placement des instructions de chargement de patch La troisième passe insère des instructions de chargement de patch `scifi.ldp` dans le code. Comme décrit dans la section 3.3, un patch est nécessaire pour tous les prédécesseurs d'un bloc de base sauf un. Dans SCI-FI, nous avons fait le choix de déclencher les mises à jour uniquement sur les branchements pris. Nous choisissons donc de placer une instruction `scifi.ldp` dans chaque prédécesseur qui ne précède pas immédiatement le bloc de base en mémoire lorsque le nombre de prédécesseurs est supérieur à 1. Lorsqu'un bloc de base n'a qu'un prédécesseur, il n'est pas nécessaire d'ajouter une instruction de chargement de patch. Une instruction de chargement de patch est aussi placée avant chaque appel de fonction et avant tous les retours d'une même fonction sauf un.

Les boucles nécessitent une attention particulière lors du placement des instructions de chargement de patch. En effet, notre règle de placement des instructions `scifi.ldp` peut échouer à briser les dépendances circulaires de signature créées par les boucles. Dans ce cas, il est nécessaire d'ajouter une instruction `scifi.ldp` supplémentaire dans un des blocs de base de la boucle. Un exemple d'une boucle nécessitant une instruction de chargement de patch supplémentaire est donné en figure 4.5. La boucle est composée des blocs de base $\mathtt{B_2}$ et $\mathtt{B_3}$. Le point d'entrée de la boucle est $\mathtt{B_2}$ et son IV est S_2, car $\mathtt{B_2}$ est le prédécesseur placé juste avant dans la mémoire. Cependant, S_2 dépend de IV_2 et donc de S_3 : $S_3 = f(B_3, IV_3)$ et $IV_3 = S_2 = f(B_2, S_3)$. Il apparait donc une dépendance circulaire entre IV_3 et S_3, ce qui empêche le calcul d'une signature pour ce CFG. Une instruction `scifi.ldp` est ajoutée arbitrairement dans $\mathtt{B_3}$ même si cela semble inutile au premier abord puisque $\mathtt{B_2}$ n'a qu'un seul prédécesseur. Cela permet d'ajouter un degré de liberté dans le calcul de la signature afin de trouver une solution aux

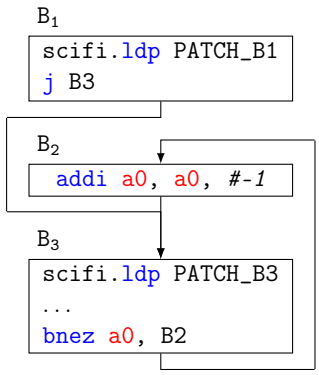

FIGURE 4.5 – Exemple d'une boucle nécessitant une instruction `scifi.ldp` supplémentaire

équations précédentes : $S_3 = F(B_3, IV_3 \oplus \text{PATCH\_B3})$.

Cette passe désactive l'utilisation de *tailcall* pour les fonctions avec plusieurs points de retour. Un *tailcall* est une instruction particulière utilisée lorsque la dernière instruction d'un bloc de base de retour d'une fonction est un appel à une autre fonction. Le *tailcall* permet alors de déléguer le retour de la fonction appelante à la fonction appelée et ainsi d'exécuter une seule séquence de retour de fonction pour les deux fonctions. Les blocs de base suivant le site d'appel de la fonction appelante attendent la signature de sortie de la fonction appelante comme IV. Cependant, avec un *tailcall*, l'IV de ces blocs de base pourra être la signature de sortie de la fonction appelée par le *tailcall*. Pour être compatible avec GPSA, il serait nécessaire que les fonctions se terminant par un *tailcall* et les fonctions appelées par les *tailcall* partagent la même signature de sortie. Cette solution augmenterait considérablement la surface d'attaque sur les retours de fonction, car plusieurs fonctions partageraient la même signature de sortie. Nous décidons donc de supprimer les *tailcall* lorsqu'une fonction a plusieurs point de retour. Dans le cas où la fonction possède un unique point de retour, celui-ci peut être un *tailcall* puisque ce sera toujours la fonction appelée par le *tailcall* qui sera responsable du retour.

Lors de cette passe, les instructions `scifi.ldp` émises n'ont pas encore de décalage d'adresse affecté et les patchs ne sont pas encore calculés. Ces opérations sont réalisées après l'édition des liens par le générateur de signatures.

Placement des instructions de vérification La dernière passe associée à SCI-FI permet de remplacer les instructions de flot de contrôle par des instructions de vérification. Plusieurs stratégies sont possibles, par exemple remplacer toutes les instructions de flot de contrôle par des instructions de vérification ou placer des instructions de vérification aux retours de fonction. Nous avons choisi de placer des instructions de vérifications dans les fonctions indiquées comme sensibles par le code source. Pour cela, un attribut est ajouté dans le langage C pour indiquer au compilateur les fonctions dans lesquelles il faut insérer des instructions de vérification : `__attribute__((scifi_secured))`. Pour ces fonctions, nous décidons de remplacer toutes les instructions de flot de contrôle par des instructions de vérification. Cette passe ajoute également un emplacement libre après l'instruction de vérification pour la signature de référence. Comme pour les patchs, les signatures de référence sont calculées par le générateur de signature après l'édition des liens. L'emplacement réservé pour la signature de référence est ensuite mis à jour.

Un autre choix est de ne placer une vérification que lors du retour de la fonction ou encore d'utiliser un mot clé pour insérer des vérifications à des points clés à l'intérieur des fonctions.

Exemple de code généré pour SCI-FI Nous présentons dans le listing 4.2 le code généré pour SCI-FI par LLVM à partir de la fonction `memcpy` du langage C (listing 4.1). Les instructions `scifi.ldp` sont placées au début des blocs de base. Cela permet d'éviter les cycles de gel dû à l'attente du chargement du patch lors des transferts de flot de contrôle afin de faire la mise à jour de la signature. De plus, les instructions de transfert de flot de contrôle sont remplacées par des instructions de vérification. Les emplacements de signature de référence sont réservés par la pseudo instruction `scifi.signature`. Ces emplacements ainsi que le décalage d'adresse des instructions `scifi.ldp` seront remplies après l'édition des liens par le générateur de signatures présenté dans la section 4.4.4.

Listing 4.1 – Fonction `memcpy` du langage C

```
void*  memcpy(
  void* dest,
  void* src,
  uint32_t n)
__attribute__((scifi_secured))
{
  uint8_t* bdest = (uint8_t*)dest;
  uint8_t* bsrc = (uint8_t*)src;
  for(uint32_t i = 0; i < n; i++)
  {
    cdest[i] = csrc[i];
  }
  return dest;
}
```

Listing 4.2 – Séquence d'instructions assembleur RISC-V générée par LLVM pour SCI-FI à partir du listing 4.1

```
memcpy:
  scifi.ldp   0(patch_base)
  mv    t1, a0
  scifi.beqz  a2, Ltmp0
  scifi.signature
Ltmp1:
  scifi.ldp   0(patch_base)
  lb    t2, 0(a1)
  sb    t2, 0(t1)
  addi  a2, a2, -1
  addi  t1, t1, 1
  addi  a1, a1, 1
  scifi.bnez  a2, Ltmp1
  scifi.signature
Ltmp0:
  scifi.ret
  scifi.signature
```

4.4.2 Générations des dispatchers

Dans le chapitre 3, nous avons présenté l'élimination des appels de fonction indirects à la compilation à l'aide de dispatchers. Comme présenté en section 4.4.1, le compilateur remplace les appels de fonctions indirects par des appels directs à des dispatchers, mais ne génère pas le code des dispatchers. La raison principale est que pour construire la liste des candidats de chaque branchement indirect, SCI-FI utilise des classes d'équivalence construites par analyse statique. Or pour réaliser cette analyse statique, il est nécessaire d'avoir une vue d'ensemble du programme avec toutes les fonctions qui pourraient avoir été compilées séparément dans plusieurs unités de compilation. Il est donc nécessaire d'accéder à l'ensemble des objets de compilation pour construire les classes d'équivalence. Cette opération doit être réalisée une fois tous

les objets de compilation émis et avant l'édition des liens. En pratique, cette opération pourrait être implémentée comme une optimisation lors de l'édition des liens (*Link Time Optimization*).

Nous utilisons les informations de debug émises par le compilateur pour construire les classes d'équivalences à partir des prototypes de fonction, par exemple les fonctions de type `int(f)(int, int)`. Ensuite, chaque appel à un dispatcher est associé à une classe d'équivalence à partir du type associé au registre qui contient l'adresse de destination du branchement indirect original. Il est alors possible de croiser ces informations pour créer les dispatchers associés à chaque appel de fonction indirect.

Chaque dispatcher est associé à une classe d'équivalence, mais aussi au registre qui contient l'adresse de la fonction cible. Il est possible d'optimiser le nombre de dispatchers pour n'en avoir plus qu'un seul par classe en utilisant un préambule pour mettre l'adresse de la fonction cible dans un registre défini indépendant du site d'appel. D'autres optimisations sont possibles telles que remplacer les dispatchers par un appel de fonction direct lorsque la classe d'équivalence ne contient qu'une seule fonction. Cependant, ces optimisations ne sont pas mises en œuvre dans le générateur de dispatchers.

4.4.3 Bibliothèque C Newlib

Nous utilisons Newlib comme bibliothèque standard du langage C, qui est destinée à des applications légères et souvent embarquées, ce qui correspond aux systèmes visés par SCI-FI. Comme il est nécessaire de calculer la signature tout au long de l'exécution du programme, il est nécessaire d'ajouter les instructions de chargement de patch dans la bibliothèque C. Pour cela, nous la compilons avec notre version modifiée de LLVM/clang. Contrairement aux instructions de chargement de patch qui doivent être ajoutées dans tout le programme, les instructions de vérification peuvent être placées arbitrairement dans le code. Nous décidons de ne pas ajouter d'instruction de vérification dans la bibliothèque C. Toutefois, l'ajout d'instruction de vérification est réalisable en ajoutant l'attribut `__attibute__((scifi_secured))` aux fonctions souhaitées.

4.4.4 Générations des signatures

Lors de l'émission du code par le compilateur, les adresses des patchs, leur valeur et les signatures de référence ne sont pas connues. Leur calcul et placement en mémoire sont réalisés par le générateur de signatures après l'édition des liens, une fois le placement du code en mémoire fixé.

Le générateur de signatures commence par extraire le CFG du programme à partir des instructions. C'est à ce moment que les instructions de chargement de patch et les instructions de vérification sont identifiées. Ensuite un algorithme récursif parcourt le CFG en commençant par le point d'entrée du programme. Pour le premier bloc de base, un IV est fixé en accord avec l'IV fixé lors de la réinitialisation du circuit, cette valeur peut être fixée lors de la conception ou alors venir d'un signal externe au processeur. Un modèle des signaux l'étage decode du CV32E40P permet de calculer le pipeline state associé à chaque instruction du bloc de base. Il est alors possible de calculer les signatures de chaque instruction dans le bloc de base à partir de l'IV et des pipeline states. Pour les blocs de base suivants, l'IV est fixé par le bloc de base

placé juste avant dans la mémoire de code.

Il peut arriver que lors du traitement du bloc de base suivant, l'IV de celui-ci ne soit pas encore calculé. C'est par exemple le cas quand il y a une mise à jour entre le bloc de base suivant et le bloc de base précédent. Dans ce cas, le CFG est parcouru de prédécesseur en prédécesseur (sens inverse de l'exécution) plutôt que successeur par successeur (sens de l'exécution) pour résoudre successivement les signatures jusqu'à obtenir la signature (et donc l'IV) manquant. Pour éviter les boucles infinies lors du parcours du CFG, chaque bloc de base est marqué après avoir été traité. Une fois toutes les signatures de référence générées, les patchs sont calculés en utilisant u^{-1} entre la signature du bloc prédécesseur et l'IV du bloc successeur.

Enfin, le générateur de signature remplit les emplacements réservés aux signatures de référence. Il place tous les patchs dans une nouvelle section de code `.patchs` et complète les décalages d'adresse de chaque instruction de chargement de patch. Il est aussi nécessaire d'ajouter l'adresse de base de la section `.patchs` pour la charger dans le CSR au démarrage du processeur.

4.5 Analyse de sécurité

Nous proposons dans cette section une analyse de la sécurité apportée par SCI-FI. Nous reprenons le modèle de menace présenté en section 3.2. Nous rappelons que nous considérons uniquement les attaques par injection de fautes sur le chemin d'instructions. Le chemin de données est supposé protégé par un mécanisme dédié et les attaques logiques sont considérées comme inexploitables.

L'attaquant peut injecter deux types de fautes dans la mémoire ou dans la logique du processeur du système ciblé :

— Une faute avec un contrôle complet sur quelques bits (typiquement moins de 8) tels que [Colombier et al. 2022] ;

— Une faute qui modifie plusieurs bits sans contrôle sur la valeur de la faute.

L'attaquant peut avoir recours à l'injection de fautes multiples dans le temps ou dans l'espace.

4.5.1 Vérification du pipeline state

Pour nos implémentations de SCI-FI, nous construisons le pipeline state à partir d'une analyse manuelle. Comme expliqué en section 4.3, pour limiter la taille du pipeline state, nous sélectionnons le nombre minimum de signaux permettant d'assurer l'intégrité du code, du flot de contrôle et de l'exécution. Cette analyse est sujette à des erreurs, et il n'y a aucune garantie que notre sélection permette de capturer toutes les fautes impactant ces propriétés de sécurité. Dans le cadre de sa thèse, Simon Tollec étudie la génération automatisée de modèles formels pour l'analyse de vulnérabilités d'un code s'exécutant sur un processeur décrit au niveau RTL, en présence de fautes. L'objectif est de réaliser des vérifications formelles de propriétés de sécurité sur un système complet (matériel et logiciel).

Nous avons collaboré afin de montrer l'utilité de l'approche qu'il propose et en même temps pour valider notre contre-mesure SCI-FI. L'objectif de la vérification menée est de montrer que le pipeline state capture bien tous les signaux de contrôle nécessaires à la protection de l'intégrité du code, du flot de contrôle et de l'exécution. Pour cela, nous considérons deux versions du programme `VerifyPIN` selon les options de compilation. La propriété de vulnérabilité à satisfaire

est “existe-t-il une faute qui ne modifie pas le pipeline state et qui permet une authentification”. En cas de satisfaisabilité, cela signifie que le pipeline state ne couvre pas tous les signaux pilotant l'exécution des instructions. En cas de non satisfaisabilité, SCI-FI via son pipeline state protège bien le programme.

Les résultats montrent que le pipeline state a bien été construit [Tollec et al. 2022]. Il montre aussi que même un code sans contre-mesure logicielle est bien protégé contre les fautes dans la microarchitecture. Ainsi, pour les deux versions du programme `VerifyPIN` considérées, SCI-FI élimine toutes les vulnérabilités : leurs nombres allant de 59 à 189 sans SCI-FI ; elles sont toutes détectées lors de l'exécution de ce programme sur le CV32E40P protégé par SCI-FI. Ces travaux permettent une première validation de la bonne construction du pipeline state dans le cas du programme `VerifyPIN`. Néanmoins, ce programme ne fait pas apparaitre toutes les structures de contrôle et toutes les instructions supportées par SCI-FI, et il est donc nécessaire de mener d'autres études pour compléter ces résultats.

4.5.2 Module d'intégrité du code et du flot de contrôle

Le module CCFI protège les étages avant du pipeline du processeur à sécuriser. Pour cela, la technique GPSA calcule une signature à partir de deux fonctions : la fonction de signature et la fonction de mise à jour. Nous étudions ici les propriétés de sécurité, présentées en section 3.4.3, des fonctions que nous avons retenues pour nos implémentations de SCI-FI.

CBC-MAC/Prince possède la propriété de résistance aux collisions grâce à l'utilisation d'une primitive de chiffrement par bloc. En effet, pour trouver une collision, il est nécessaire d'inverser la primitive de chiffrement et donc de connaitre la clé secrète. La robustesse d'une construction CBC-MAC dépend de la robustesse de la primitive. Pour Prince, la clé secrète a une taille de 128 bits tandis que les blocs manipulés ont une taille de 64 bits. Pour une clé inconnue de l'attaquant, forger un nouveau bloc de base dont la signature est identique au bloc de base original est réalisable avec une probabilité de $\frac{1}{2^{64}}$, soit un niveau de sécurité de 64 bits. Cependant, uniquement 32 bits de la signature sont comparés à la signature de référence lors des vérifications, comme expliqué dans la section 4.3. Il est donc possible de créer une collision lors de la vérification avec une probabilité de $\frac{1}{2^{32}}$, soit un niveau de sécurité de 32 bits. Toutefois, cette attaque serait détectée lors de la vérification suivante puisque les 32 bits non utilisés pour la vérification sont utilisés pour le calcul des signatures suivantes. Un tel niveau de sécurité est considéré comme trop faible par une partie de la communauté de recherche en cryptographie (niveau inférieur à 80 bits de sécurité [Aumasson 2019]). Cependant, dans notre modèle d'attaquant, le nombre de bits de sécurité ne représente pas le nombre d'opérations à effectuer par un ordinateur, mais plutôt le nombre d'injections de fautes différentes et réussites nécessaires pour créer une collision. Dans ce contexte, 32 bits de sécurité nous semblent suffisants pour protéger un système en considérant qu'il faudrait faire $2^{31} \simeq 2 \times 10^9$ injections de fautes pour avoir une probabilité de réussir l'attaque de 50 %. De plus, il est difficile de déterminer si l'attaque a permis d'obtenir une collision, car le registre de signature n'est pas accessible à l'attaquant.

Pour les CBC-MACs, plusieurs restrictions sont normalement imposées pour garantir leur sécurité. Premièrement, les CBC-MACs ne doivent pas être utilisés avec des messages de taille

variable. En effet, un attaquant peut faire une attaque par concaténation en forgeant un nouveau triplet (IV, bloc de base, signature) valide à partir de triplets (IV, bloc de base, signature) existants. À partir des blocs de base (IV_1, BB_1, S_1) et (IV_2, BB_2, S_2), il est possible de concaténer BB_1 et BB_2 composés des instructions $I_{i.0}$ à $I_{i.n}$ pour créer un nouveau triplet (IV_1, BB_3, S_2) :

$$BB_3 = BB_1 \parallel S_1 \oplus I_{2.0} \oplus IV \parallel I_{2.1} \parallel \ldots \parallel I_{2.n} \tag{4.1}$$

Une solution courante à ce problème est de fixer la tailles de blocs de base comme dans SOFIA [Clercq et al. 2016], néanmoins cette solution augmente fortement la taille du code. Une autre solution est d'utiliser un protocole dans lequel le nombre d'instructions est indiqué en début de bloc de base. Cependant, cette attaque par concaténation est peu réaliste en pratique. En effet, pour forger un nouveau bloc de base, l'attaquant doit concaténer deux blocs de base existants, ce qui est difficile en utilisant uniquement des injections de faute. Même en relâchant notre modèle de menace et en considérant un attaquant capable de reprogrammer la mémoire de code, seul les blocs de base précédés et suivis par une vérification de signature sont utilisables pour construire l'attaque, car il est nécessaire de connaitre l'IV et la signature de chaque bloc. Enfin, la combinaison d'une signature, d'un IV et de la première instruction d'un bloc de base donnera une instruction valide avec une probabilité de 4,4 % pour le jeu RV32I en raison de la faible densité de ce jeu d'instructions (189 169 666 instructions valides). La seule exception est lorsque la signature et l'IV sont égaux, mais dans ce cas, les blocs de base se suivent légitimement dans le CFG. C'est pourquoi SCI-FI autorise les blocs de base de taille variable.

Une seconde restriction concernant les CBC-MACs est l'utilisation d'IVs variables, car il est possible de créer un nouveau triplet (IV, bloc de base, signature) en inversant les mêmes bits dans la première instruction du bloc de base et dans l'IV. Cependant, cela nécessite l'utilisation d'une double injection de fautes très précise avec la première sur le registre de signature pour modifier l'IV et la seconde sur le pipeline state de la première instruction du bloc de base ce qui encore une fois rend l'attaque peu réaliste.

Notre seconde implémentation utilise un CRC32 pour le calcul de la signature. Contrairement à l'implémentation CBC-MAC/Prince, la fonction de signature CRC32 ne permet pas d'assurer l'authenticité du code. De plus, le CRC32 est une fonction inversible, ce qui signifie qu'un attaquant peut forger de nouveaux blocs de base dont la signature est identique à celle du bloc de base à attaquer. Les CRC32 sont conçus pour détecter un minimum d'inversions de bits en fonction de la taille du message. Nous évaluons plusieurs CRC32 connus pour leur capacité de détection [Koopman 2002]. Pour cela, nous cherchons l'ensemble des vecteurs de collisions, c'est-à-dire un vecteur de 64 bits à injecter dans le pipeline state pour créer une collision, dont le poids de Hamming (nombre de bits égaux à 1) est inférieur à 10. Notons que dans notre modèle de menace, nous considérons un attaquant capable d'injecter une faute avec un contrôle précis sur huit bits maximum. Ici, une injection de fautes est représentée par le OU exclusif entre un mot binaire M et la représentation binaire de la faute E : $M' = M \oplus E$. Dans notre cas, M est une instruction et E est le vecteur de collision. Nous cherchons donc $E \neq 0$ tel que $\text{CRC32}(IV, M \oplus E) = \text{CRC32}(IV, M)$. Le CRC32 étant une fonction linéaire par rapport au OU exclusif, $\text{CRC32}(IV, M \oplus E) = \text{CRC32}(IV, M) \oplus \text{CRC32}(IV, E)$. Cette

TABLEAU 4.1 – Poids de Hamming pour différents CRC32 en fonction du nombre d'instructions (t) entre les deux injections de fautes

CRC32	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
0xFA567D89	8	8	8	8	8	8
0xD419CC15	8	8	8	8	8	8
0xBA0DC66B	8	8	8	8	8	8
0xB49C1C96	8	8	6	8	8	8
0x992C1A4C	8	8	6	6	8	8
0x90022004	8	8	8	6	8	8
0x8F6E37A0	6	8	6	6	6	6
0x82608EDB	7	8	6	8	8	8
0x80108400	5	5	5	6	6	6

propriété permet de limiter notre évaluation à la recherche des vecteurs de collision respectant la propriété suivante : $\text{CRC32}(IV, E) = 0$. Nous testons les vecteurs de collision en les triant par poids de Hamming afin de trouver le nombre minimum d'inversions de bit nécessaire pour créer une collision avec une injection de faute unique dans le pipeline state. Cette méthode peut être généralisée en chainant le calcul des CRC32s comme pour le calcul de la signature : $\text{CRC32}(\text{CRC32}(\text{CRC32}(IV, E_1), E_2 \ldots), E_n) = 0$. Ainsi, nous cherchons les vecteurs de collisions pour une injection de faute double pour des blocs de base de 2 à 7 instructions. Entre les deux injections, nous utilisons des vecteurs de collisions $E_i = 0$ pour représenter une instruction non fautée et uniquement E_1 et E_n sont différents de 0. Le tableau 4.1 présente les résultats pour les CRC32 évalués provenant de [KOOPMAN 2002]. Une seconde évaluation des deux meilleurs candidats pour des blocs de base plus longs nous montre que le CRC32 `0xFA567D89` permet de détecter 8 inversions de bit pour des séquences allant de 1 à 30 instructions. Cela permet donc de se protéger d'un attaquant capable de réaliser des injections de fautes multiples sur 8 bits, ce qui est au-delà de l'état de l'art tel que décrit dans le chapitre 2.

La fonction de mise à jour peut aussi être source de vulnérabilité. Elle permet de directement manipuler le registre de signature en utilisant le patch préalablement chargé. Cependant, la mise à jour est effectuée entre deux calculs de signature et est donc équivalent à fauter le registre de signature directement. Pour les deux implémentations, la robustesse de la fonction de mise à jour dépend de la robustesse de la fonction de signature. Un attaquant pourrait forger un nouveau patch pour détourner le flot de contrôle, cependant cette attaque nécessite à la fois de modifier le patch et de détourner le flot de contrôle et nécessite donc l'utilisation d'injection de fautes multiples. Cette attaque est peu réaliste sans un contrôle précis des bits modifiés lors de l'injection de faute, ce qui encore une fois sort de notre modèle de menace La propriété de confidentialité du code et des métadonnées de SCI-FI permettrait d'éliminer complètement cette menace.

Le dernier point critique du module CCFI est la vérification de signature. La technique GPSA et le chaînage des signatures permettent à ce module d'être robuste à une faute qui empêcherait la vérification. En effet, la faute est toujours propagée dans les signatures suivantes et détectée

lors de la prochaine vérification. Une fois une faute capturée par le mécanisme de signature, il est nécessaire de fauter toutes les vérifications ultérieures pour contourner la détection.

Une attaque possible serait de détourner le flot de contrôle vers une séquence d'instructions dans laquelle il n'y a aucune vérification de signature. Dans ce cas, le module CCFI n'est plus capable de détecter aucune faute. Le niveau de sécurité assuré par SCI-FI dépend du délai de détection des fautes dans la signature et donc de l'intervalle entre deux instructions de vérification. Il est possible de déterminer l'intervalle maximum entre deux instructions de vérification et même de le contraindre en ajoutant des instructions de vérification additionnelles lors de la compilation. À partir de la connaissance de l'intervalle maximum entre deux instructions de vérification, il est possible d'utiliser un *watchdog* pour compter le nombre d'instructions entre deux vérifications. Ainsi, lorsque la valeur du compteur du watchdog dépasse la valeur maximum fixée, par exemple lors de la compilation, une exception peut être levée pour interrompre le programme. Cette solution permet de détecter toutes attaques détournant le flot de contrôle en dehors du programme protégé par des vérifications de signature.

4.5.3 Module d'intégrité des signaux de contrôle

Le module CSI couvre tous les signaux de contrôle issus du décodage et transmis de l'étage decode aux étages suivants. Une attaque sur le pipeline state après l'étage decode ne sera pas détectée uniquement si la faute cible simultanément un signal et son duplicata dans le module CSI. Toutefois, il est possible d'utiliser des méthodes de duplication plus robustes, par exemple la duplication complémentaire pour répondre à ce problème. Nous pensons que l'utilisation de méthode de duplication plus complexe aurait un impact limité sur l'augmentation de la surface de circuit en raison de la faible quantité de logique à protéger.

Un problème courant avec les mécanismes de détection d'erreur est l'encodage du résultat de la comparaison sur un seul bit [Schilling et al. 2018]. Il est alors possible d'utiliser une seule faute pour inverser le résultat de la comparaison. Une technique commune pour prévenir ce type de vulnérabilité est l'utilisation d'encodage différentiel sur deux bits. Ainsi, uniquement deux valeurs parmi les quatre combinaisons possibles sont valides. De plus, il est nécessaire d'injecter deux fautes différentes pour changer le résultat de la comparaison. Il est important de noter que ce type de mécanisme peut aussi être utilisé pour protéger le résultat d'autres comparaisons sensibles, par exemple pour les branchements, et pour le résultat de la vérification du module CCFI.

4.5.4 Intégrité du flot de contrôle

Une particularité de SCI-FI est l'utilisation de l'élimination des branchements indirects en plus de la technique GPSA pour protéger l'intégrité du flot de contrôle. À l'aide de ces deux techniques, SCI-FI réduit considérablement le nombre de transitions illégitimes dans le CFG qui ne seraient pas détectés. Dans notre modèle de menace, une attaque sur le flot de contrôle serait envisageable en modifiant le registre contenant l'adresse d'une fonction dans un dispatcher. Cependant, cette attaque viserait l'intégrité des données, qui est supposée être protégée par une protection complémentaire.

Nous considérons ici des attaques qui sortent de notre modèle de menace. La sécurité des

Tableau 4.2 – Nombre de dispatchers et nombre de classes d'équivalence pour les programmes avec branchements indirects

Programme	Nombre de dispatchers	Nombre de classes d'équivalence	Tailles des classes	Fonctions illegitimes
picojpeg (`Os`)	1	1	1	0
picojpeg (`O2`)	2	1	1	0
sglib-combined (`Os`)	2	2	1 et 3	1 et 3
sglib-combined (`O2`)	1	1	3	3
wikisort (`Os`)	9	2	1 et 9	0 et 0
wikisort (`O2`)	3	2	1 et 9	0 et 0

branchements indirects dépend de la précision de la construction des classes d'équivalence et plus précisément de la taille des classes d'équivalence. Le tableau 4.2 donne la taille des classes d'équivalence pour les programmes considérés pour l'évaluation logicielle (Section 4.6). Ce tableau rapporte également le nombre de dispatchers ajoutés pour chaque programme, le nombre de classes d'équivalence, et le nombre de fonctions non légitimes par classe. Le nombre de dispatchers dépend du nombre d'appel de fonction indirects dans le programme original. La taille des classes d'équivalence dépend du nombre de fonctions partageant le même prototype dans le code source. Dans les programmes évalués, le nombre de dispatchers ne dépasse pas deux et la plus grande classe contient neuf éléments, ce qui réduit considérablement la surface d'attaque sur les branchements indirects. Il est possible que plusieurs dispatchers soient associés à la même classe d'équivalence lorsque le registre contenant l'adresse de la destination varie. C'est par exemple le cas pour `wikisort` et pour `sglib-combined` avec le niveau d'optimisation `Os` pour lesquels il y a plus de dispatchers que de classes d'équivalence. Enfin, les fonctions non légitimes sont des fonctions qui ne sont pas atteignables, depuis un site d'appel de fonction donné, sans attaque. Dans le contexte des classes d'équivalence, c'est une fonction qui partage le même prototype que les fonctions légitimes, mais qui ne peut pas être appelée depuis un site d'appel de fonction indirect donné. Parmi les trois programmes, `sglib-combined` est le seul qui possède des fonctions non légitimes dans ces classes d'équivalence. De plus, ses classes d'équivalence sont composées uniquement de fonctions non légitimes. L'analyse statique qui identifie les classes d'équivalence est insensible à certains contextes dans le code source, notamment elle n'est pas capable d'identifier quand un pointeur de fonction contient `NULL`. En conséquence, une à deux classes d'équivalence inutiles sont générées pour ce programme. Ce problème peut être résolu par une analyse manuelle du code source et des métriques rapportées par le générateur de dispatchers pour sélectionner les dispatchers à inclure dans le programme final.

En conclusion, même face à un attaquant capable de modifier les données, SCI-FI laisse uniquement une faible surface d'attaque sur le flot de contrôle. De plus SCI-FI permet de se protéger partiellement contre les attaques logicielles telles que le ROP ou le JOP.

Tableau 4.3 – Évaluation du surcoût matériel de SCI-FI dans la technologie GF-22FDX FDSOI à 400 MHz

	Surface (kGE)	Surcoût (%)
Base	51,8	-
CRC32	55,2	6,5
CBC-MAC/Prince	64,2	23,8

4.6 Évaluation de l'impact sur les performances

Nous évaluons les performances de SCI-FI autant au niveau matériel que logiciel. Pour cela, nous réalisons une synthèse de SCI-FI pour un *Application-Specific Integrated Circuit* (ASIC) pour estimer le surcoût matériel. Nous utilisons la suite de test Embench-IoT pour évaluer le surcoût logiciel à l'aide d'un outil de simulation de circuit [*Embench™ : Open Benchmarks for Embedded Platforms* 2021]. Cette section présente les résultats obtenus.

4.6.1 Synthèse ASIC

Nous réalisons la synthèse du CV32E40P dans la technologie GF-22FDX FDSOI avec l'outil Design Compiler de Synopsys. La synthèse du processeur de référence occupe pour notre configuration une surface de 51,8 kGE pour une fréquence de 400 MHz. Pour rappel, la porte équivalente ou *Gate Equivalent* en anglais (GE) est une unité qui représente la complexité d'un circuit indépendamment de la technologie utilisée. C'est le rapport entre la surface du circuit et la plus petite cellule disponible dans la technologie utilisée. Pour la technologie GF-22FDX FDSOI, la porte logique de référence est une porte *nand* à deux entrées (SC8T_ND2X0P5_CSC24SL) dont la surface est de 0,199 68 µm^2.

Nous réalisons une synthèse pour chaque implémentation de SCI-FI. L'implémentation CRC32 occupe 55,2 kGE soit une augmentation de la surface du circuit de 6,5% par rapport à la version de référence. L'implémentation CBC-MAC/Prince occupe 64,2 kGE soit 23,8% de plus que la version de référence et 16,2% de plus que l'implémentation CRC32 de SCI-FI. Ces résultats sont récapitulés dans le tableau 4.3.

Il est important de noter que ces deux implémentations de SCI-FI n'augmente pas le chemin critique du circuit pour une fréquence de 400 MHz et permettent donc de ne pas réduire la fréquence de fonctionnement du circuit. Une étude plus poussée serait nécessaire pour déterminer précisément la fréquence de fonctionnement maximale et l'impact de SCI-FI sur cette fréquence.

4.6.2 Environnement d'évaluation logiciel

Pour évaluer les performances logicielles de SCI-FI, nous utilisons Verilator, un outil de simulation au niveau transfert de registre (RTL) avec une précision temporelle de l'ordre du cycle d'horloge [*Verilator* 2022]. Verilator permet de transformer une représentation RTL d'un circuit, par exemple dans le langage SystemVerilog, en un modèle C++.

Le système simulé est composé du CV32E40P, de 4 Mo de mémoire et de périphériques dont notamment un port série. Nous ajoutons également deux adresses mémoires qui, lorsqu'elles font l'objet d'une transaction, indiquent le cycle de début et de fin du code exécuté. Cela permet de mesurer le temps d'exécution des fonctions de la suite de test.

Une autre option pour réaliser l'évaluation serait d'implémenter SCI-FI sur un circuit de type *Field-Programmable Gate Array* (FPGA). Un FPGA permet un temps d'évaluation beaucoup plus rapide puisque le système est directement câblé dans un circuit reprogrammable. Cependant, certaines fonctionnalités telles que la simulation d'injection de fautes sont plus difficiles à mettre en œuvre avec un FPGA. Bien que finalement peu exploré dans cette thèse par manque de temps, nous avons conservé l'utilisation d'un simulateur dans un objectif ultérieur de simulation d'injection de faute.

4.6.3 Évaluation logicielle

Nous utilisons la suite de test Embench-IoT pour évaluer les performances logicielles de l'implémentation CRC32 de SCI-FI [*Embench™ : Open Benchmarks for Embedded Platforms* 2021]. Cette suite de test est conçue pour évaluer des performances en taille de code et en temps d'exécution des systèmes embarqués sans système d'exploitation. Embench-IoT est composé de 19 programmes qui représentent différents profils de flot de contrôle, de calcul et d'accès mémoire. Parmi ces 19 programmes, 3 contiennent des branchements indirects.

Pour l'évaluation, tous les programmes sont compilés avec la chaine de compilation de SCI-FI en utilisant les niveaux d'optimisations `Os` qui minimise la taille du code et `O2` qui minimise le temps d'exécution. Nous utilisons la bibliothèque C Newlib et le support logiciel pour les multiplications et les nombres flottants de LLVM. Le code inutile est supprimé à l'aide des options `-ffunction-sections` et `-gc-sections`. Enfin, les programmes de Embench-IoT sont modifiés pour insérer les instructions de vérification uniquement dans les fonctions principales et non dans les fonctions utilisées pour l'instrumentation du programme ou dans la bibliothèque C.

La taille de code est mesurée en considérant les sections `.text` et `.patchs`. Le coût sur la taille de code est donc une borne supérieure du coût de SCI-FI puisque notre évaluation ne prend pas en compte la mémoire occupée par les données. Le temps d'exécution est le nombre de cycles nécessaires pour exécuter la fonction principale dans chaque programme de la suite de test.

Les résultats obtenus pour l'évaluation logicielle sont présentés dans le tableau 4.4. Pour chaque programme la taille du code est indiqué en nombre d'octets et en coût relatif à la version de référence. Le tableau 4.4 indique également le nombre de patchs et de signatures de référence présent dans chaque programme. Il est important de noter que pour la taille du code, un patch implique un surcoût mémoire de huit octets (quatre octets pour l'instruction de chargement et quatre octets pour le patch) et la signature de référence représente quatre octets. Le temps d'exécution est indiqué en nombre de cycles ainsi qu'en surcoût relatif à la version de référence. Ces résultats sont également rapportés dans les figures 4.6 et 4.7 dans lesquelles la moyenne géométrique est indiquée par une ligne de tirets.

TABLEAU 4.4 – Résultats de la suite Embench-IoT avec la taille (en octets, et par rapport à la version non protégée), le nombre de signatures et de patchs ainsi que le temps d'exécution (en cycles CPU, et par rapport à la version non protégée)

Programme	O2				Os			
	Taille	Signatures	Patches	Temps d'exéc.	Taille	Signatures	Patches	Temps d'exéc.
aha-mont64	6204 (×1.30)	124	106	75335 (×1.38)	4720 (×1.28)	92	70	67461 (×1.18)
crc32	824 (×1.34)	8	21	380997 (×1.21)	856 (×1.35)	9	21	381006 (×1.21)
cubic	97460 (×1.16)	98	1611	14787617 (×1.16)	96920 (×1.16)	95	1608	14802517 (×1.16)
edn	3564 (×1.29)	53	63	1157814 (×1.22)	3944 (×1.25)	53	64	1157585 (×1.22)
huffbench	4028 (×1.39)	90	90	382404 (×1.20)	3548 (×1.33)	61	73	383163 (×1.16)
matmult-int	3376 (×1.26)	29	70	1119255 (×1.21)	2588 (×1.23)	10	54	1200057 (×1.21)
minver	21472 (×1.24)	56	489	151685 (×1.18)	21404 (×1.24)	59	474	178843 (×1.17)
nbody	20576 (×1.20)	52	404	55767175 (×1.18)	19944 (×1.20)	42	387	55780098 (×1.18)
nettle-aes	5600 (×1.14)	46	63	136279 (×1.10)	5512 (×1.14)	44	59	136447 (×1.10)
nettle-sha256	7864 (×1.07)	39	44	9573 (×1.03)	7720 (×1.08)	46	46	10239 (×1.03)
nsichneu	25204 (×1.45)	654	565	3693 (×1.44)	25152 (×1.45)	648	546	3685 (×1.44)
qrduino	21840 (×1.39)	554	455	1605197 (×1.18)	18304 (×1.37)	448	357	1610103 (×1.16)
slre	7604 (×1.55)	242	211	0	6760 (×1.52)	214	168	45480 (×1.17)
st	21964 (×1.20)	58	420	6618952 (×1.17)	21764 (×1.19)	44	409	6626187 (×1.17)
statemate	8956 (×1.29)	203	141	1573 (×1.09)	9116 (×1.28)	198	138	1672 (×1.08)
ud	3456 (×1.25)	39	62	23674 (×1.16)	3232 (×1.27)	37	62	24125 (×1.16)
picojpeg	31116 (×1.47)	881	665	2403701 (×1.21)	22548 (×1.49)	642	485	2424312 (×1.16)
sglib-combined	8332 (×1.47)	197	219	409637 (×1.18)	8736 (×1.46)	206	214	466871 (×1.31)
wikisort	32340 (×1.30)	354	647	28465027 (×1.38)	31956 (×1.29)	318	625	24817089 (×1.20)
Moyennes	**17462 (×1.30)**	**99**	**334**	**5973662 (×1.20)**	**16564 (×1.29)**	**86**	**308**	**5795628 (×1.18)**

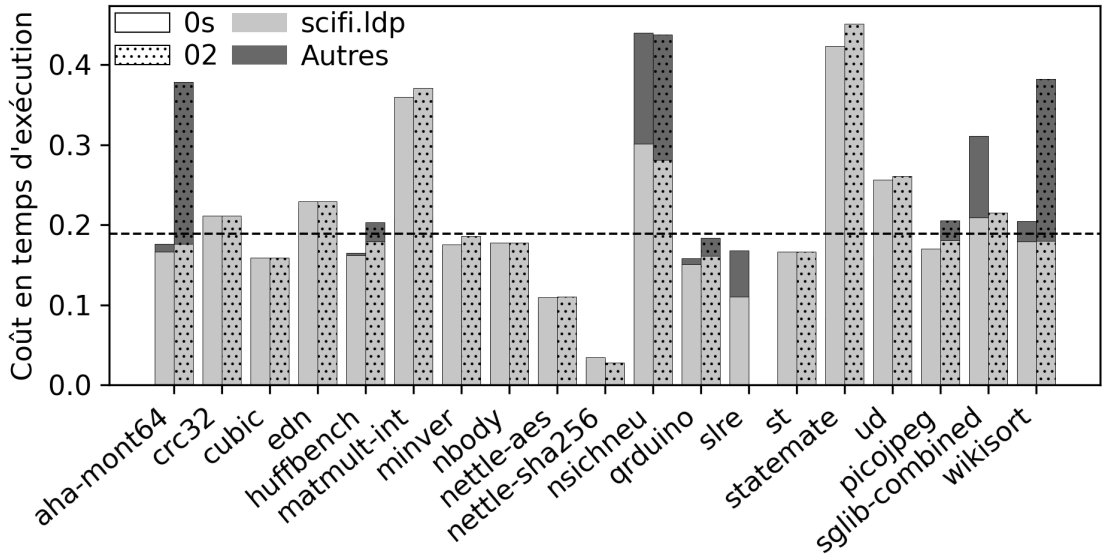

FIGURE 4.6 – Surcoût en temps d'exécution pour la suite de test Embench-IoT avec les niveaux d'optimisation `Os` et `O2` avec la moyenne géométrique en ligne de tirets.

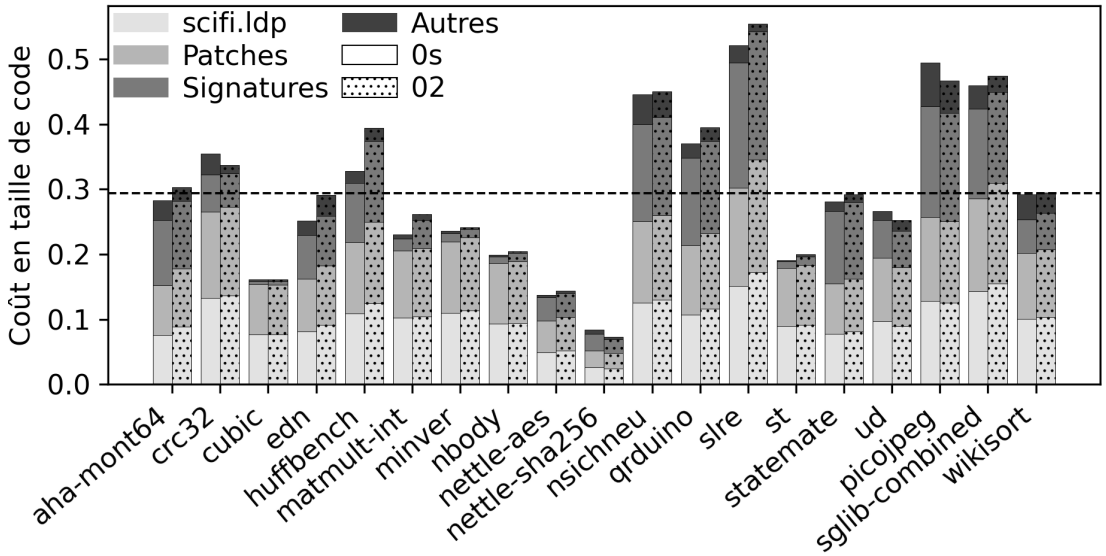

FIGURE 4.7 – Surcoût en taille de code pour la suite de test Embench-IoT avec les niveaux d'optimisation `Os` et `O2` avec la moyenne géométrique en ligne de tirets.

Le surcoût en temps d'exécution varie entre 2,5 % et 44,0 % avec une moyenne géométrique de 18,4 %. Pour la taille de code, le surcoût est compris entre 7,3 % et 55,4 % avec une moyenne géométrique de 29,4 %. On observe sur la figure 4.6 que pour la majorité des programmes, la différence de surcoût entre les niveaux d'optimisations `Os` et `O2` est négligeable. Néanmoins, les programmes `aha-mont64`, `sglib-combined` et `wikisort` présentent une forte différence en temps d'exécution suivant le niveau d'optimisation. Cette différence est due à des séquences d'instructions comprenant un `scifi.ldp` suivie d'une instruction de flot de contrôle dans un nid de boucle. Ces séquences d'instructions provoquent un cycle de gel avant le transfert de flot de contrôle pour attendre le chargement du patch pour la mise à jour. Cela provoque un grand nombre de cycles de gel, car ces séquences d'instructions apparaissent dans des nids de boucle. Cela explique donc la différence en temps d'exécution.

L'évaluation montre que l'instruction de chargement de patch (`scifi.ldp`) est responsable de la majorité du surcoût de SCI-FI. Le nombre de vérifications de signature est dépendant du nombre de transferts de flot de contrôle dans les fonctions annotées avec `scifi_secured` alors que tout le code est instrumenté avec des `scifi.ldp`.

Outre les instructions dédiées à SCI-FI, d'autres transformations ajoutent des instructions et augmentent donc la taille du code et le temps d'exécution. Ces instructions sont représentées dans la catégorie *Autres* sur la figure 4.7. Premièrement, l'élimination des dépendances aux frontières des blocs de base ajoute des instructions `nop` dans le code. De plus, l'ajout d'instructions `scifi.ldp` et d'emplacement pour les signatures de référence augmente la distance entre les branchements et leurs cibles. Il est alors possible que le décalage d'adresse nécessaire pour atteindre la cible d'un branchement ne soit plus encodable dans la partie réservée à cet effet. Dans ce cas, le branchement est transformé en une nouvelle séquence d'instructions qui permet d'atteindre une adresse plus éloignée. Cette transformation contribue également au surcoût *Autres*.

Parmi les programmes de la suite Embench-IoT, trois utilisent des appels de fonction indi-

TABLEAU 4.5 – Coût en taille de code en octets (contribution au coût global) et nombre de patchs et de signatures ajoutés par les dispatchers

Programme	Total	Patches	Signatures
picojpeg (`Os`)	84 (1.1%)	4	2
picojpeg (`O2`)	168 (1.7%)	4	4
sglib-combined (`Os`)	256 (9.3%)	8	6
sglib-combined (`O2`)	172 (6.4%)	6	4
wikisort (`Os`)	1108 (15.3%)	34	26
wikisort (`O2`)	604 (8.2%)	22	14

rects. Le tableau 4.5 rapporte le surcoût induit par l'utilisation de dispatchers pour éliminer les branchements indirects. Dans le pire cas, les dispatchers contribuent à 15,3 % de l'augmentation de la taille du code pour `wikisort` avec le niveau d'optimisation `Os`, soit une augmentation de 4 % de la taille du code par rapport à la version de référence non protégée. Pour tous les autres programmes, les dispatchers contribuent à moins de 10 % de l'augmentation de la taille de code. Le coût des dispatchers sur la taille de code pourrait encore être diminué en implémentant les optimisations décrites dans la section 4.4.2. Ces résultats montrent que l'utilisation de dispatchers pour éliminer les branchements indirects est une solution réaliste dans un contexte embarqué pour limiter le nombre de blocs de base qui partagent le même IV avec GPSA au prix d'une légère augmentation de la taille de code.

Nous observons que le niveau d'optimisation du compilateur a un effet modéré sur le surcoût en taille de code et en temps d'exécution. Cependant, il existe une grande variation entre les différents programmes évalués. Ces variations sont dues aux différents types de programmes compris dans la suite de test (flot de contrôle, calcul ou accès mémoire). Par exemple, un bloc de base de petite taille sera plus impacté par l'instrumentation de SCI-FI. Le surcoût maximum est de 400% pour un bloc de base avec une seule instruction, car SCI-FI ajoute trois mots mémoire : une instruction `scifi.ldp`, le patch associé et la signature de référence. Cependant, nos résultats montrent que le surcoût moyen est bien inférieur, car les blocs de base sont composés de plusieurs instructions. Certaines optimisations peuvent aider à réduire le surcoût de SCI-FI en augmentant la taille des blocs de base, par exemple le déroulage des boucles.

Les résultats présentés dans cette section concernent l'implémentation CRC32 de SCI-FI. Pour l'implémentation CBC-MAC/Prince, il est possible d'estimer le surcoût logiciel à partir des résultats précédents. Les deux implémentations calculent les signatures en un seul cycle d'horloge. De plus, les signatures de référence sont composées du même nombre de bits (32). La différence principale se situe au niveau des mises à jour. L'implémentation CBC-MAC/Prince travaille avec un état interne de 64 bits (taille des blocs pour Prince). Il est donc nécessaire de faire des mises à jour sur 64 bits, ce qui double l'empreinte mémoire des patchs par rapport à l'implémentation CRC32. SCI-FI impliquerait donc un surcoût moyen de 39% pour le temps d'exécution et de 50% pour la taille du code avec l'implémentation CBC-MAC/Prince. Néanmoins, cette implémentation permet d'utiliser le maximum des capacités de SCI-FI au niveau de la sécurité et assure ainsi la propriété d'authenticité du code.

Une solution pour limiter le coût d'utilisation d'une fonction de signature cryptographique est de choisir une fonction de signature qui travaille sur une taille de bloc plus petite. Par exemple, l'algorithme de chiffrement par bloc Simon, dans sa version 32 bits, serait un bon candidat pour remplacer Prince dans la construction du CBC-MAC. Pour conserver un pipeline state sur 64 bits, il est possible d'utiliser le CRC32 comme fonction de compression du pipeline state puis CBC-MAC/Simon pour chainer les signatures. Cette solution permettrait d'obtenir des performances logicielles similaires à l'implémentation CRC32 et la propriété d'authenticité du code au prix d'une légère augmentation de la surface du circuit.

4.7 Conclusion

Dans ce chapitre, nous avons présenté deux implémentations de notre contre-mesure SCI-FI. Nous proposons une implémentation utilisant une fonction de signature cryptographique qui permet d'atteindre un haut niveau de sécurité. Nous proposons également une implémentation plus légère utilisant une fonction de signature de type code correcteur d'erreurs au détriment de la propriété d'authenticité du code.

Nous avons réalisé une analyse de sécurité de nos deux implémentations. La version cryptographique, CBC-MAC/Prince, de SCI-FI est robuste face à un attaquant et demande au moins 2×10^9 injections de fautes pour réussir une attaque non détectée avec une probabilité supérieure à 50 %. La version CRC32 de SCI-FI permet de détecter jusqu'à huit inversions de bit, ce qui en fait une contre-mesure robuste face à l'attaquant de l'état de l'art. De plus, le mécanisme de vérification renforce la sécurité du module CCFI, car toute faute, une fois injectée, sera détectée par les vérifications successives. Le module CSI est lui robuste face à une injection de faute simple et peut être modifié pour contraindre le type de fautes multiples à injecter. SCI-FI améliore également la sécurité contre les attaques visant le flot de contrôle grâce à l'utilisation de dispatchers pour éliminer les branchements indirects. L'analyse sur nos programmes de test montre que cette solution laisse une surface d'attaque petite et difficile à exploiter.

Nous avons évalué le coût matériel de SCI-FI par la synthèse de notre processeur dans un flot de conception ASIC. Nos résultats montrent que même pour une implémentation cryptographique, l'impact de SCI-FI sur la surface du processeur à sécuriser est limité à quelques dizaines de milliers de portes équivalentes, ce qui est faible en regard d'un système complet comprenant de la mémoire. L'implémentation CRC permet de limiter l'augmentation de la taille du circuit à quelques milliers de portes équivalentes. Cela montre que le choix de la fonction de signature permet de jouer sur le compromis entre le niveau de sécurité assuré par SCI-FI et l'impact sur les performances du système. Dans les deux cas, la fonction de signature n'augmente pas le chemin critique de notre processeur. Pour l'évaluation logicielle, nos résultats montrent que SCI-FI est compatible avec des programmes typiques des systèmes embarqués tels que ceux présents dans Embench-IoT. Les performances logicielles de SCI-FI varient grandement d'un programme à l'autre en fonction de la taille des blocs de base, notamment dans les boucles et de la forme du CFG. Cependant, les programmes avec un flot de contrôle indépendant des données tels que `nettle-aes` et `nettle-sha256` sont faiblement impactés par SCI-FI. De plus, l'analyse sur les performances montre que l'élimination des branchements indirects à la compilation impacte faiblement la taille de code et le temps d'exécution des programmes ainsi sécurisés.

En conclusion, ce chapitre montre que notre contre-mesure SCI-FI est utilisable en pratique pour protéger des systèmes embarqués tout en limitant la dégradation des performances matérielles et logicielles.

Chapitre 5

Conclusion

Sommaire du présent chapitre

5.1 Conclusion générale

Aujourd'hui les systèmes numériques sont omniprésents dans notre environnement quotidien, ils peuvent manipuler des données sensibles qui permettent de nous identifier, d'assurer la confidentialité de nos échanges ou de réaliser d'autres actions critiques au niveau de la sécurité. Les attaques matérielles et plus particulièrement les attaques par injection de fautes remettent en cause les hypothèses de sécurité classiques sur ces systèmes numériques. Notamment, il est possible de perturber l'exécution d'un programme sur un circuit en compromettant les propriétés d'intégrité du code, ou du flot de contrôle, ou en perturbant la microarchitecture du circuit.

Dans cette thèse, nous nous sommes intéressés à la protection du chemin d'instructions d'un système numérique. Nous avons remarqué que les propriétés d'intégrité du code et du flot de contrôle ne sont pas suffisantes pour assurer la protection de l'intégralité du chemin d'instruction. En effet, une perturbation de la logique de contrôle dans la microarchitecture peut modifier le comportement d'une instruction lors de son exécution. Nous avons donc proposé une nouvelle propriété de sécurité que nous appelons *intégrité d'exécution* et qui garantit la bonne exécution des instructions dans la microarchitecture. Nous avons ensuite étudié la conception puis l'implémentation d'une contre-mesure qui assure simultanément les propriétés d'intégrité du code, du flot de contrôle et d'exécution couvrant ainsi l'intégralité du chemin d'instruction. Une telle contre-mesure doit être la plus légère possible et doit pouvoir fonctionner en cas d'interruptions et en présence de branchements indirects.

Dans le chapitre 3 nous avons décrit le principe de fonctionnement de SCI-FI, la contre-mesure mixte matérielle et logicielle proposée en réponse à notre problématique. SCI-FI est composé de deux modules matériels qui s'articulent autour du *pipeline state* pour protéger l'intégralité du chemin d'instruction. Le pipeline state est un vecteur de bits construit à partir

des signaux de contrôle émis par l'étage decode. Le premier module utilise la technique de signature généralisée aux chemins GPSA pour assurer l'intégrité du code, du flot de contrôle et d'exécution pour les étages avant du pipeline. Contrairement aux implémentations classiques de GPSA qui calculent la signature à partir de l'encodage des instructions, ce module utilise le pipeline state. Cela permet notamment d'assurer l'intégrité des signaux de contrôle émis par l'étage decode et donc d'assurer une partie de l'intégrité d'exécution. Le second module réutilise le pipeline state pour assurer la propagation des signaux qui le composent. Ce module permet de compléter la couverture de l'intégrité d'exécution dans le pipeline tout en limitant la surface du circuit à dupliquer. En effet, seule la logique de contrôle est dupliquée, ce qui permet des implémentations légères en comparaison aux techniques de duplication classiques pour les unités de calcul. Dans la conception de SCI-FI, nous avons pris en compte plusieurs fonctionnalités de pipeline classiques. SCI-FI est totalement compatible avec la prédiction de branchement à l'aide de modifications de la génération du code dans la chaîne de compilation. SCI-FI supporte et sécurise également les interruptions, nécessaires pour traiter les événements externes au processeur de manière asynchrone et pour la construction de systèmes embarqués. Enfin, SCI-FI supporte les appels de fonction indirects en les éliminant lors d'une passe de transformation du code dans la chaîne de compilation. Cela renforce encore plus le niveau de sécurité apporté par SCI-FI contre des attaques sur le flot de contrôle. Bien que nous ayons étudié SCI-FI dans le cadre d'un pipeline court et peu complexe, une étude préliminaire montre que SCI-FI pourrait être adaptée à des pipelines plus longs et plus complexes sans pour autant augmenter considérablement son coût.

Nous avons proposé deux implémentations de SCI-FI dans le chapitre 4. Nous avons intégré SCI-FI dans le processeur RISC-V 32 bits CV32E40P avec un pipeline de quatre étages. Pour montrer les différents compromis possibles avec SCI-FI, nous avons réalisé une implémentation utilisant une fonction de signature cryptographique et une autre utilisant un CRC32. Pour le support logiciel, nous avons modifié le back-end de la chaîne de compilation LLVM et développé les outils nécessaires pour avoir une chaîne de développement logicielle complète. Nous avons analysé le niveau de sécurité assuré par nos implémentations de SCI-FI. Une collaboration avec Tollec et al. a permis de faire une première validation de la construction de notre pipeline state et de montrer que SCI-FI permet d'éliminer toutes les vulnérabilités dans le cas d'étude choisi. Une analyse plus théorique montre qu'il est nécessaire de faire 2×10^9 injections de fautes pour avoir plus de 50 % de chance de réussir une attaque non détectée sur le mécanisme de signature pour l'implémentation cryptographique. Pour l'implémentation CRC32, il est nécessaire de contrôler précisément au moins huit bits de la signature, ce qui aujourd'hui est au-delà des capacités d'un attaquant à l'état de l'art. Enfin, notre analyse montre également que la protection du flot de contrôle offert par SCI-FI réduit grandement la surface d'attaque du système, même face à un attaquant utilisant des attaques logicielles classiques telles que le ROP ou le JOP. Nous avons également évalué l'impact de SCI-FI sur les performances du système. Une synthèse de SCI-FI dans un ASIC indique que la surface matérielle augmente de 23,8 % et de 6,5 % respectivement pour les implémentations cryptographique et CRC32. Ces résultats sont prometteurs et montrent que SCI-FI impacte légèrement la surface occupée par le processeur qui est elle-même petite en comparaison à la surface d'un circuit comprenant de la mémoire et des périphériques. Pour l'évaluation logicielle, nous avons exécuté avec SCI-FI différents programmes représentatifs des applications embarquées. Nos résultats montrent que

l'impact de SCI-FI sur les performances logicielles varient fortement suivant le type d'application. SCI-FI augmente en moyenne le temps d'exécution de 18,4 % et la taille du code de 29,4 %, ce qui est comparable ou inférieure aux contre-mesures logicielles tout en offrant un niveau de sécurité supérieur grâce à la propriété d'intégrité d'exécution.

5.2 Perspectives

SCI-FI s'appuie entièrement sur le pipeline state pour assurer la protection du chemin d'instruction. Pour nos implémentations, nous avons construit le pipeline state à partir d'une analyse manuelle du pipeline à sécuriser. Une première amélioration de SCI-FI serait l'automatisation de la construction du pipeline state. Cela permettrait entre autre d'adapter plus facilement SCI-FI d'un pipeline à un autre et pourrait ainsi faciliter l'adoption d'une telle contre-mesure.

Nous avons dans nos travaux étudié deux fonctions de signature pour les implémentations de SCI-FI, l'une cryptographique, l'autre basée sur un code détecteur d'erreurs. Une amélioration possible pour SCI-FI serait de construire une fonction de signature qui soit en même temps robuste contre la recherche de vecteur de collision (faute à injecter pour créer une collision dans la signature) et qui impose un poids de Hamming minimum pour les vecteurs de collisions. Cela permettrait d'assurer un niveau de sécurité borné même lorsque le secret associé à la fonction cryptographique est compromis, par exemple par une attaque par canal auxiliaire.

SCI-FI repose sur une technique de redondance matérielle pour assurer l'intégrité des signaux de contrôle dans les étages arrière du pipeline. Nous avons proposé une implémentation utilisant la duplication. La duplication double la surface de la logique à protéger alors qu'elle ne permet de détecter que les injections de faute simple. Une injection de fautes double permet de contourner ce type de protection. Il serait intéressant d'étudier des mécanismes différents, par exemple à base de codes détecteurs d'erreurs pour remplacer le mécanisme de duplication.

Nous avons évalué le niveau de sécurité apporté par nos implémentations de SCI-FI à l'aide d'une analyse théorique. Cependant, cette analyse est réalisée manuellement et elle peut être sujette à des erreurs ou des oublis. Il est nécessaire pour garantir le niveau de sécurité de SCI-FI de valider le fonctionnement des différents éléments qui la compose. La vérification formelle du matériel et du logiciel semble être une solution intéressante pour réaliser cette vérification. C'est notamment ce qui a commencé à être fait dans le cadre d'une collaboration avec Simon Tollec dont la thèse porte sur la génération automatisée de modèles formels pour l'analyse de vulnérabilité d'un code s'exécutant sur un processeur décrit au niveau RTL, en présence de fautes. De tels modèles permettent de mettre en évidence de potentiels chemins d'attaque, ce qui permet de guider les futures améliorations à apporter à SCI-FI.

Dans ces travaux, nous nous sommes concentrés sur la sécurisation de pipelines simples, représentatifs des systèmes embarqués, qui sont les cibles privilégiées des attaques par injection de fautes. Néanmoins, nous avons vu dans le chapitre 2 que les attaques par injection de fautes peuvent aujourd'hui être réalisées sans accès physique, par exemple avec un accès logique tel qu'une connexion réseau. Il est probable que dans le futur, ces attaques puissent cibler des serveurs qui utilisent des pipelines bien plus complexes que dans les systèmes embarqués. Dans un premier temps, SCI-FI pourrait être portée sur des pipelines plus complexes, par exemple avec de l'exécution dans le désordre. Il faudrait également faire une étude du modèle de menace pour ces nouvelles cibles afin de proposer des contre-mesures qui pourraient être plus adaptées,

par exemple en considérant les attaques logicielles.

Les travaux de cette thèse se concentrent sur la sécurisation du chemin d'instruction contre les attaques par injection de fautes. Les attaques par canal auxiliaire ne sont pas considérées. Pourtant, elles pourraient être utilisées contre SCI-FI, par exemple pour retrouver la clé de la primitive cryptographique et réduire la complexité d'une attaque sur la collision de la signature. De futurs travaux pourraient étudier la gestion et la sécurisation de la clé utilisée dans SCI-FI. Sur un autre axe, SCI-FI peut impacter les attaques par canal auxiliaire sur l'application exécutée. Il serait intéressant d'étudier l'empreinte de SCI-FI sur ces attaques pour voir si leur complexité augmente ou diminue.

Nous n'avons pas non plus considéré les attaques par injection de fautes sur le chemin de données. Les données en mémoire sont souvent protégées par des codes détecteurs d'erreurs. Cependant, tout comme pour le chemin d'instruction, la microarchitecture est vulnérable à des attaques sur le chemin de données. Une piste d'étude pour des travaux futurs serait de combiner SCI-FI avec un mécanisme de protection du chemin de données dans la microarchitecture.

Enfin, la confidentialité du code est une propriété importante pour se protéger contre la rétro ingénierie, notamment utilisée lors de la construction des attaques. Des travaux récents combinent la confidentialité du code avec les propriétés d'intégrité du code et du flot de contrôle [Clercq et al. 2016 ; Werner et al. 2018 ; Savry et al. 2020]. Nous avons également commencé à étudier comment intégrer la propriété de confidentialité du code dans SCI-FI. Ces travaux font actuellement l'objet d'une demande de brevet.

Publications et communications personnelles

Conférence avec acte

Chamelot, T., D. Courousse et K. Heydemann (2022). «SCI-FI : Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks». In : DATE.

Présentations

Chamelot, T., D. Courousse et K. Heydemann (2021). «SCI-FI : Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks». In : JAIF.

Chamelot, T., D. Courousse et K. Heydemann (2022). «SCI-FI : Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks». In : PHISIC

Poster

Chamelot, T., D. Courousse et K. Heydemann (2022). «SCI-FI : Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks». In : RISC-V Week

Article de journal soumis

Chamelot, T., D. Courousse et K. Heydemann (2022). «MAFIA : Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks».

Demande de brevet déposée

Chamelot, T., D. Courousse et K. Heydemann (01/09/2022). «Procédé d'exécution d'un code machine par un calculateur». Numéro de demande 2208801

Bibliographie

Abadi, M., M. Budiu, Ú. Erlingsson et J. Ligatti (2009). « Control-Flow Integrity Principles, Implementations, and Applications ». In : *ACM TISSEC* 1.

Agoyan, M., J.-M. Dutertre, D. Naccache, B. Robisson et A. Tria (2010). « When Clocks Fail : On Critical Paths and Clock Faults ». In : *CARDIS*.

Amiel, F., C. Clavier et M. Tunstall (2006). « Fault Analysis of DPA-Resistant Algorithms ». In : *FDTC*.

Anceau, S., P. Bleuet, J. Clédière, L. Maingault, J.-l. Rainard et R. Tucoulou (2017). « Nanofocused X-Ray Beam to Reprogram Secure Circuits ». In : *CHES*.

Arora, D., S. Ravi, A. Raghunathan et N. K. Jha (2006). « Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors ». In : *IEEE Trans. on VLSI Systems* 12.

Arthur, W., B. Mehne, R. Das et T. Austin (2015). « Getting in Control of Your Control Flow with Control-Data Isolation ». In : *CGO*.

Aumasson, J.-P. (2019). « Too Much Crypto ». In : *Cryptology ePrint Archive, Paper 2019/1492*.

Baksi, A., D. Saha et S. Sarkar (2019). « To Infect Or Not To Infect : A Critical Analysis Of Infective Countermeasures In Fault Attacks ». In : *Cryptology ePrint Archive, Report 2019/355*.

Barenghi, A., L. Breveglieri et I. Koren (2010). « Countermeasures Against Fault Attacks on Software Implemented AES : Effectiveness and Cost ». In : *WESS*.

Barry, T. (2017). « Sécurisation à la compilation de logiciels contre les attaques en fautes ». Thèse de doct. Ecole des mines de Saint-Etienne.

El-Baze, D., J.-B. Rigaud et P. Maurine (2016). « A Fully-Digital EM Pulse Detector ». In : *DATE*.

Berzati, A., C. Canovas-Dumas et L. Goubin (2012). « A Survey of Differential Fault Analysis Against Classical RSA Implementations ». In : *Fault Analysis in Cryptography*. Springer.

Biham, E. et A. Shamir (1997). « Differential Fault Analysis of Secret Key Cryptosystems ». In : *CRYPTO '97*.

Binder, D., E. C. Smith et A. B. Holman (1975). « Satellite Anomalies from Galactic Cosmic Rays ». In : *IEEE Trans. Nucl. Sci* 6.

Bletsch, T., X. Jiang, V. W. Freeh et Z. Liang (2011). « Jump-Oriented Programming : A New Class of Code-Reuse Attack ». In : *ASIACCS*.

Boneh, D., R. A. DeMillo et R. J. Lipton (1997). « On the Importance of Eliminating Errors in Cryptographic Computations ». In : *J. Cryptol.* 2.

Borghoff, J. et al. (2012). « PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications ». In : *ASIACRYPT*.

Burow, N. et al. (2017). « Control-Flow Integrity : Precision, Security, and Performance ». In : *ACM Comput. Surv.* 1.

Claudepierre, L., P.-Y. Péneau, D. Hardy et E. Rohou (2021). « TRAITOR : A Low-Cost Evaluation Platform for Multifault Injection ». In : *ASIACCS*.

Clavier, C. (2012). « Attacking Block Ciphers ». In : *Fault Analysis in Cryptography*. Springer.

Clercq, R. de et al. (2016). « SOFIA : Software and Control Flow Integrity Architecture ». In : *DATE*.

Collberg, C. et J. Nagra (2009). *Surreptitious Software : Obfuscation, Watermarking, and Tamperproofing for Software Protection : Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

Colombier, B., A. Menu, J.-M. Dutertre, P.-A. Moellic, J.-B. Rigaud et J.-L. Danger (2019). « Laser-Induced Single-bit Faults in Flash Memory : Instructions Corruption on a 32-Bit Microcontroller ». In : *HOST*.

Colombier, B. et al. (2022). « Multi-Spot Laser Fault Injection Setup : New Possibilities for Fault Injection Attacks ». In : *CARDIS*.

Costan, V. et S. Devadas (2016). « Intel SGX Explained ». In : *Cryptology ePrint Archive, Paper 2016/086*.

Criteria, C. (2017). *Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 5*.

Danger, J.-L. et al. (2020). « Processor Anchor to Increase the Robustness Against Fault Injection and Cyber Attacks ». In : *COSADE*.

Dobraunig, C., M. Eichlseder, T. Korak, S. Mangard, F. Mendel et R. Primas (2018). « SIFA : Exploiting Ineffective Fault Inductions on Symmetric Cryptography ». In : *TCHES*.

Dumont, M., M. Lisart et P. Maurine (2021). « Modeling and Simulating Electromagnetic Fault Injection ». In : *TCAD* 4.

Dutertre, J.-M. et al. (2014). « Laser Attacks on Integrated Circuits : From CMOS to FD-SOI ». In : *DTIS*.

Embench™ : Open Benchmarks for Embedded Platforms (2021). url : `https://github.com/embench/embench-iot` (visité le 15/07/2021).

Endo, S., Y. Li, N. Homma, K. Sakiyama, K. Ohta et T. Aoki (2012). « An Efficient Countermeasure against Fault Sensitivity Analysis Using Configurable Delay Blocks ». In : *FDTC*.

Gandolfi, K., C. Mourtel et F. Olivier (2001). « Electromagnetic Analysis : Concrete Results ». In : *CHES*.

Glamocanin, O., D. G. Mahmoud, F. Regazzoni et M. Stojilovic (2021). « Shared FPGAs and the Holy Grail : Protections against Side-Channel and Fault Attacks ». In : *DATE*.

GliGli et Tiros (2012). *The Xbox 360 Reset Glitch Hack*. wiki. url : `https://free60project.github.io/wiki/Reset_Glitch_Hack.html` (visité le 29/11/2019).

Gravellier, J., J.-M. Dutertre, Y. Teglia et P. L. Moundi (2021). « FaultLine : Software-Based Fault Injection on Memory Transfers ». In : *HOST*.

Grycel, J. et P. Schaumont (2021). « SimpliFI : Hardware Simulation of Embedded Software Fault Attacks ». In : *Cryptography* 2.

Guillen, O. M., M. Gruber et F. De Santis (2017). « Low-Cost Setup for Localized Semi-invasive Optical Fault Injection Attacks ». In : *COSADE*.

He, W., J. Breier et S. Bhasin (2016). « Cheap and Cheerful : A Low-Cost Digital Sensor for Detecting Laser Fault Injection Attacks ». In : *SPACE*.

Hutter, M. et J.-M. Schmidt (2013). « The Temperature Side-Channel and Heating Fault Attacks ». In : *CARDIS*.

Karaklajić, D., J.-M. Schmidt et I. Verbauwhede (2013). « Hardware Designer's Guide to Fault Attacks ». In : *IEEE VLSI* 12.

Kim, S. et A. K. Somani (2001). « On-Line Integrity Monitoring of Microprocessor Control Logic ». In : *Microelectronics Journal*.

Kim, Y. et al. (2014). « Flipping Bits in Memory without Accessing Them : An Experimental Study of DRAM Disturbance Errors ». In : *ISCA*.

Koopman, P. (2002). « 32-Bit Cyclic Redundancy Codes for Internet Applications ». In : *DSN*.

Korak, T. et M. Hoefler (2014). « On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms ». In : *FDTC*.

Lac, B., A. Canteaut, J. J. A. Fournier et R. Sirdey (2018). « Thwarting Fault Attacks against Lightweight Cryptography Using SIMD Instructions ». In : *ISCAS*.

Lalande, J.-F., K. Heydemann et P. Berthomé (2014). « Software Countermeasures for Control Flow Integrity of Smart Card C Codes ». In : *ESORICS*.

Laurent, J., V. Beroulle, C. Deleuze, F. Pebay-Peyroula et A. Papadimitriou (2019). « Cross-Layer Analysis of Software Fault Models and Countermeasures against Hardware Fault Attacks in a RISC-V Processor ». In : *Microprocessors and Microsystems*.

Maene, P. et I. Verbauwhede (2016). « Single-Cycle Implementations of Block Ciphers ». In : *LightSec*.

Mangard, S., E. Oswald et T. Popp (2007). *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. 1st. Springer.

Menu, A., J.-M. Dutertre, O. Potin, J.-B. Rigaud et J.-L. Danger (2020). « Experimental Analysis of the Electromagnetic Instruction Skip Fault Model ». In : *DTIS*.

Moro, N. (2014). « Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués ». Thèse de doct. Université Pierre et Marie Curie.

Moro, N., A. Dehbaoui, K. Heydemann, B. Robisson et E. Encrenaz (2013). « Electromagnetic Fault Injection : Towards a Fault Model on a 32-Bit Microcontroller ». In : *FDTC*. arXiv : `1402.6421`.

Murdock, K., D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss et F. Piessens (2020). « Plundervolt : Software-based Fault Injection Attacks against Intel SGX ». In : *S&P*.

O'Flynn, C. (2016). « Fault Injection Using Crowbars on Embedded Systems ». In : *IACR Cryptology ePrint Arch.*

Ordas, S., L. Guillaume-Sage et P. Maurine (2017). « Electromagnetic Fault Injection : The Curse of Flip-Flops ». In : *JCEN* 3.

Ozdoganoglu, H., T. Vijaykumar, C. Brodley, B. Kuperman et A. Jalote (2006). « SmashGuard : A Hardware Solution to Prevent Security Attacks on the Function Return Address ». In : *IEEE Transactions on Computers* 10.

Pan, J., F. Zhang, K. Ren et S. Bhasin (2019). « One Fault Is All It Needs : Breaking Higher-Order Masking with Persistent Fault Analysis ». In : *DATE*.

Patrick, C., B. Yuce, N. F. Ghalaty et P. Schaumont (2017). « Lightweight Fault Attack Resistance in Software Using Intra-instruction Redundancy ». In : *SAC*.

Piret, G. et J.-J. Quisquater (2003). « A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad ». In : *CHES*.

Proy, J., K. Heydemann, A. Berzati et A. Cohen (2017). « Compiler-Assisted Loop Hardening Against Fault Attacks ». In : *TACO* 4.

Proy, J., K. Heydemann, F. Majéric, A. Cohen et A. Berzati (2019). « Studying EM Pulse Effects on Superscalar Microarchitectures at ISA Level ».

Qiu, P., D. Wang, Y. Lyu et G. Qu (2019). « VoltJockey : Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies ». In : *SIGSAC*.

Reis, G., J. Chang, N. Vachharajani, R. Rangan et D. August (2005). « SWIFT : Software Implemented Fault Tolerance ». In : *CGO*.

Roscian, C., A. Sarafianos, J.-M. Dutertre et A. Tria (2013). « Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells ». In : *FDTC*.

San Pedro, M., M. Soos et S. Guilley (2011). « FIRE : Fault Injection for Reverse Engineering ». In : *WISTP*.

Savry, O., M. El-Majihi et T. Hiscock (2020). « Confidaent : Control FLow Protection with Instruction and Data Authenticated Encryption ». In : *DSD*.

Schilling, R., M. Werner et S. Mangard (2018). « Securing Conditional Branches in the Presence of Fault Attacks ». In : *DATE*.

Seaborn, M. et D. Dullien (2015). « Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges ». In : *Black Hat*.

Shacham, H. (2007). « The Geometry of Innocent Flesh on the Bone : Return-into-libc without Function Calls (on the X86) ». In : *CCS*.

Sharma, A. et S. K. Sahay (2014). « Evolution and Detection of Polymorphic and Metamorphic Malwares : A Survey ». In : *IJCA*.

Skorobogatov, S. P. et R. J. Anderson (2003). « Optical Fault Induction Attacks ». In : *CHES*.

Sohier, T., J. P. Michel, S. Borel, J. C. Souriau, G. Simon et A. Tria (2022). « Evaluation of Magnetoimpedance in Narrow NiFe/Al/NiFe Thin Films for Secured Packaging ». In : *IEEE Transactions on Magnetics* 2.

Tang, A., S. Sethumadhavan et S. Stolfo (2017). « CLKSCREW : Exposing the Perils of Security-Oblivious Energy Management ». In : *USENIX*.

Timmers, N., A. Spruyt et M. Witteman (2016). « Controlling PC on ARM Using Fault Injection ». In : *FDTC*.

Tollec, S., M. Asavoae, D. Courousse, K. Heydemann et M. Jan (2022). « Exploration of Fault Effects on Formal RISC-V Microarchitecture Models ». In : *FDTC*.

Trouchkine, T., S. K. Bukasa, M. Escouteloup, R. Lashermes et G. Bouffard (2021). « Electromagnetic Fault Injection against a System-on-Chip, toward New Micro-Architectural Fault Models ». In : *JCEN*. arXiv : 1910.11566.

Van den Herrewegen, J., D. Oswald, F. D. Garcia et Q. Temeiza (2020). « Fill Your Boots : Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis ». In : *TCHES*.

Vasselle, A., H. Thiebeauld, Q. Maouhoub, A. Morisset et S. Ermeneux (2017). « Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot ». In : *FDTC*.

Verilator (2022). url : `https://www.veripool.org/verilator/` (visité le 07/09/2022).

Wang, R. et Y. Yan (2022). « A Survey of Secure Boot Schemes for Embedded Devices ». In : *ICACT*.

Waterman, A., K. Asanovic et C. Division (2019). *The RISC-V Instruction Set Manual Volume I : Unprivileged ISA, Version 20191213*.

Werner, M., T. Unterluggauer, D. Schaffenrath et S. Mangard (2018). « Sponge-Based Control-Flow Protection for IoT Devices ». In : *EuroS&P*.

Werner, M., E. Wenger et S. Mangard (2015). « Protecting the Control Flow of Embedded Processors against Fault Attacks ». In : *CARDIS*.

Wilken, K. et J. P. Shen (1990). « Continuous Signature Monitoring : Low-Cost Concurrent Detection of Processor Control Errors ». In : *TCAD* 6.

Yuce, B., P. Schaumont et M. Witteman (2018). « Fault Attacks on Secure Embedded Software : Threats, Design, and Evaluation ». In : *HaSS* 2.

Yuce, B. et al. (2017). « Analyzing the Fault Injection Sensitivity of Secure Embedded Software ». In : *TECS* 4.

Zussa, L., J.-M. Dutertre, J. Clediere et A. Tria (2013). « Power Supply Glitch Induced Faults on FPGA : An in-Depth Analysis of the Injection Mechanism ». In : *IOLTS*.

Sécurisation de l'exécution des applications contre les attaques par injection de fautes par une contre-mesure intégrée au processeur

Résumé

Les systèmes embarqués numériques sont omniprésents dans notre environnement quotidien. Ces systèmes embarqués, par leur caractère nomade, sont particulièrement sensibles aux attaques dites par injection de fautes. Par exemple, un attaquant peut injecter une perturbation physique dans un circuit électronique pour compromettre les fonctionnalités de sécurité du système. Originellement utilisées pour compromettre des systèmes cryptographiques, ces attaques permettent aujourd'hui de cibler n'importe quel type de système. Ces attaques permettent notamment de compromettre l'exécution d'un programme.

Dans ce manuscrit, nous introduisons une nouvelle propriété de sécurité pour protéger l'exécution des instructions dans la microarchitecture : *l'intégrité d'exécution*. À partir de cette propriété, nous décrivons le concept de SCI-FI, une contre-mesure qui assure la protection de l'intégralité du chemin d'instructions en assurant l'intégrité du code, du flot de contrôle et d'exécution. Pour cela, nous construisons un vecteur de bits que nous appelons *pipeline state* à partir de signaux de contrôle dans la microarchitecture. À partir du pipeline state, deux modules s'articulent pour assurer les propriétés de sécurité. Le premier module calcule une signature à partir du pipeline state assurant ainsi l'intégrité du code, du flot de contrôle et une partie de l'intégrité d'exécution. Le second module complète l'intégrité d'exécution dans la microarchitecture en utilisant un mécanisme de redondance. Nous proposons également le support et la sécurisation des branchements indirects et des interruptions, nécessaires pour la conception de systèmes embarqués.

Nous réalisons deux implémentations de SCI-FI, l'une construite sur une primitive cryptographique assurant un niveau de sécurité maximal et l'autre plus légère construite sur une fonction CRC privilégiant les performances. Pour cela, nous intégrons SCI-FI dans un processeur RISC-V 32 bits et modifions la chaîne de compilation LLVM. Nous réalisons une analyse de sécurité des différents éléments qui composent SCI-FI dans chaque implémentation. Nous montrons ainsi que SCI-FI, même avec l'implémentation privilégiant les performances, est robuste face à un attaquant disposant de moyens d'injection de fautes à l'état de l'art. Enfin, nous évaluons les performances de nos implémentations par une synthèse dans un flot de conception ASIC et par l'exécution en simulation de la suite de test Embench-IOT. Nous montrons ainsi que SCI-FI a des performances équivalentes aux contre-mesures de l'état de l'art tout en assurant une propriété de sécurité supplémentaire, l'intégrité d'exécution.

Mots clés : injection de fautes, contre-mesure, microarchitecture

Abstract

Embedded systems are ubiquitous in our everyday life. Those embedded systems, by their nomadic nature, are particularly sensitive to the so-called fault injection attacks. For example, an attacker might inject a physical perturbation in an integrated circuit to compromise the security features of the system. Originally used to compromise cryptographic systems, those attacks can now target any kind of system. Notably, those attacks enable to compromise the execution of a program.
In this manuscript, we introduce a new security property to protect the execution of instructions in the microarchitecture: *execution integrity*. From this property, we describe the concept of SCI-FI, a counter-measure that ensures the protection of the whole instruction path thanks to code, control-flow and execution integrity properties. We build SCI-FI around a bit vector that we call *pipeline state* and that is composed of microarchitecture control signals. Two modules interact around the pipeline state to ensure the security properties. The first module computes a signature from the pipeline state to ensure code and control-flow integrity and partially execution integrity. The second module completes the execution integrity support in the microarchitecture thanks to a redundancy mechanism. We also propose a solution for indirect branches and interrupts that are required to design embedded systems. We implement two versions of SCI-FI, one built around a cryptographic primitive which provides the best security level and another lighter one built around a CRC to maximize the performances. We integrate SCI-FI into a 32 bits RISC-V processor, and we modify the LLVM compiler. We analyze the security provided by our two implementations and we show that SCI-FI, even with the lightweight implementation, is robust against state-of-the-art attacker. Finally, we evaluate the performances of our implementations through an ASIC synthesis and through the execution of the benchmark suite Embench-IoT. We show that SCI-FI has comparable performances to state-of-the-art counter-measures while ensuring a new security property: execution integrity.

Keywords: fault injection, counter-measure, microarchitecture