Quantification de la sécurité des applications en présence d'attaques physiques et détection de chemins d'attaques

Jean-Baptiste Bréjon

▶ **To cite this version:**

Jean-Baptiste Bréjon. Quantification de la sécurité des applications en présence d'attaques physiques et détection de chemins d'attaques. Cryptographie et sécurité [cs.CR]. Sorbonne Université, 2020. Français. NNT : 2020SORUS275 . tel-03987599v2

HAL Id: tel-03987599

https://theses.hal.science/tel-03987599v2

Submitted on 14 Feb 2023

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

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

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunication et Électronique (Paris)

Présentée par

Jean-Baptiste BRÉJON

Pour obtenir le grade de :

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Quantification de la sécurité des applications en présence d'attaques physiques et détection de chemins d'attaques

Soutenue le 26/06/2020

Mr. Antoine Miné	Président	Sorbonne Université
Mme. Marie-Laure Potet	Rapporteur	Grenoble INP - Ensimag
Mr. Jean-Max Dutertre	Rapporteur	École des Mines de Saint-Étienne
Mme. Karine Heydemann	Directeur	Sorbonne Université
Mme. Emmanuelle Encrenaz	Co-directeur	Sorbonne Université
Mme. Patricia Mouy	Examinateur	ANSSI
Mr. Quentin Meunier	Encadrant	Sorbonne Université

Résumé

Les systèmes embarqués traitent et manipulent de plus en plus de données sensibles. Ils sont de plus en plus variés et touchent des domaines comme la santé, l'aérospatial ou encore les transports. La sécurité de ces systèmes est une préoccupation de premier ordre pour ceux qui les conçoivent. Les attaques en fautes visent à perturber l'exécution des programmes à travers la manipulation de grandeurs physiques dans l'environnement du système et permettent, par exemple, de contourner des mécanismes de sécurité ou encore de réaliser une escalade de privilèges. Pour faire face à cette menace, des contre-mesures logicielles prenant la forme d'ajout de code autour de la fonctionnalité à protéger sont déployées. Différentes analyses sont actuellement utilisées pour évaluer l'efficacité des contre-mesures une fois déployées mais, elles présentent des limitations : elles sont peu ou pas automatisées, coûteuses et selon la méthode employée elles sont limitées quant à la couverture des comportements possibles et aux types de fautes analysables.

Nous proposons une méthode d'analyse de robustesse de code binaire combinant des méthodes formelles et de l'exécution symbolique. L'analyse au niveau du code binaire permet non seulement de se placer après la compilation, qui peut altérer les contre-mesures, mais aussi de prendre en compte des éléments du code binaire invisibles à plus haut niveau. Les méthodes formelles, capables d'exhaustivité, permettent à l'analyse de considérer toutes les configurations des paramètres d'entrée. L'analyse est toutefois réalisée vis-à-vis d'un contexte symbolique, extrait par exécution symbolique, ce qui la circonscrit à des paramètres d'entrée réalistes et limite ainsi les faux positifs.

Nous avons implémenté cette méthode dans un outil, nommé `RobustB`, automatisé depuis le code source. Il supporte différents types de fautes transitoires affectant aussi bien le code que les données. De plus, nous proposons trois métriques permettant de synthétiser les résultats de l'analyse et d'aider le concepteur de contre-mesures à évaluer la sensibilité globale du code ainsi qu'au niveau de chaque instruction. Nos expériences montrent l'utilité de `RobustB` et de ses métriques pour analyser les effets de la compilation, déterminer la redondance de protections, vérifier le bon déploiement de contre-mesures insérées à la compilation, ou encore rechercher des chemins d'attaques.

Abstract

Embedded systems are processing and handling more and more sensitive data. They now come in multiple forms and affect areas such as health, aerospace or transportation. The security of these systems is now a prime concern for those who designs them. Fault attacks are indented to disrupt the execution of programs through the manipulation of physical quantities in the system environment and enable an attacker to bypass security mechanisms or achieve privilege escalation. To address this threat, software counter-measures in the form of additional code interleaving with the sensitive functionality are deployed. Various analyses are now being used to assess the efficiency of the counter-measures once deployed. Yet, those analyses have limitations : they are little or not automated, they are costly and, depending of the method used, they are limited in terms of code coverage of the possible behaviour and of faults types that can be analysed.

We propose a method to analyse the robustness of binary code combining formal methods and symbolic execution. Performing the analysis at the binary level presents two major advantages. First, it positions the analysis after compilation which can affect the counter-measures ; second, it allows to take into account information which is only visible at the binary level and which can be exploited to perform an attack. Formal methods are capable of exhaustiveness and thus allow the analysis to consider all possible configurations of inputs. The proposed analysis is nevertheless carried out with respect to a symbolic context, extracted by symbolic execution, which confines it to a realistic set of inputs and thus limits false positives.

We have implemented this method in a tool called `RobustB`. It is automated from the source code and supports various types of transient faults affecting both the code and the data. In addition, we propose three metrics synthesising the analysis results and helping the designer of counter-measures to assess the sensitivity of the code as a whole and at the granularity of an instruction. Our experiences show the usefulness of our tool and metrics to analyse and understand the effects of compilation, determine protections redundancy, check the proper deployment of counter-measures when inserted at compile time and search for attack paths.

Table des matières

Table des figures

Liste des tableaux

Chapitre

1 Introduction

Les systèmes embarqués réalisant des traitements sensibles (authentifications, paiements, ...) font partie de notre vie quotidienne depuis la démocratisation de la carte bancaire en France en 1992. Depuis quelques années, nous assistons à l'émergence d'objets connectés manipulant ou traitant des données toutes autant sensibles. Ces systèmes sont de plus en plus variés et touchent des domaines comme la santé, l'aérospatial ou encore les transports. Regroupés sous l'appellation d'Internet des Objets (Internet of Things, IoT), ils ont la particularité d'avoir la capacité de communiquer sur Internet. Ces objets sont destinés à opérer sans surveillance dans un environnement où leur accès physique n'est pas toujours restreint. Ces systèmes traitant des informations sensibles, leur sécurité est une préoccupation de premier ordre pour ceux qui les conçoivent.

La norme traitant de la sécurité de l'information [Hum06] définit trois axes jaugeant la sécurité : la confidentialité, l'intégrité et la disponibilité. La confidentialité garantit que l'information n'est visible que pour les personnes qui y sont autorisées. L'intégrité assure que l'information n'a pas été modifiée par un tiers qui n'en a pas l'autorisation. La disponibilité correspond à un taux défini d'accessibilité des informations ou d'un service.

La confidentialité repose entre autre sur le chiffrement des données réalisé par différents algorithmes tels que l'Advanced Encryption Standard (AES) ou encore RSA (du nom des ses auteurs, Rivest, Shamir et Adleman). Avec Boneh *et al.* [BDML97], l'année 1997 marque le début d'une nouvelle classe d'attaques : les attaques physiques. Cette classe d'attaques repose sur l'exploitation du matériel et non plus uniquement sur les propriétés mathématiques des algorithmes de chiffrement. Autrement dit, les garanties mathématiques apportées par les algorithmes de chiffrement ne suffisent plus, on parle d'attaques d'implémentation. On distingue deux catégories d'attaques physiques, les attaques par canaux auxiliaires et les attaques en faute. Les premières ont pour but d'obtenir des informations par l'observation de phénomènes physiques en lien avec l'exécution d'un programme (temps,

consommation, émission électromagnétique, ...). Les secondes visent à perturber l'exécution des programmes à travers la manipulation de quantités physiques dans l'environnement du système et permettent, par exemple, de contourner des mécanismes de sécurité ou encore de réaliser une escalade de privilège. Ce sont ces dernières auxquelles nous nous intéressons dans cette thèse.

Pour faire face à cette menace, des contre-mesures matérielles et logicielles sont déployées dans ces systèmes. Le choix des contre-mesures dépend des systèmes ciblés, de leurs contraintes et de leurs usages. La sécurité des systèmes est fonction de la surface de silicium disponible, des performances requises et des ressources supposées des attaquants potentiels. Des contre-mesures matérielles peuvent être déployées et prennent alors la forme de capteurs ou autres détecteurs placés dans ou autour de la puce. N'affectant pas, ou peu, les performances du système, elles ont toutefois le désavantage de ne pas être flexibles. En effet, en cas de nouvelles menaces identifiées, la partie matérielle du système ne peut être mise à jour et doit donc être repensée, produite et remplacée, processus nécessitant un lourd investissement. Ce n'est pas le cas pour les contre-mesures logicielles car ces dernières sont beaucoup moins chères à concevoir et à déployer, peuvent être mises à jour et donc être adaptées pour faire face aux nouvelles menaces.

Les contre-mesures logicielles peuvent être déployées à différents niveaux de représentation du code (source, assembleur, binaire). Au niveau source, les contre-mesures logicielles sont susceptibles d'être altérées voir totalement supprimées par les passes d'optimisation du compilateur. Elles peuvent aussi être insérées au niveau assembleur mais la perte de sémantique du programme rend leur déploiement plus difficile. Il est donc nécessaire de garantir non seulement l'efficacité d'une contre-mesure logicielle déployée mais aussi de son intégrité. C'est pourquoi, le processus de sécurisation de logiciels, partant du choix d'une contre-mesure jusqu'à son déploiement dans le produit final, inclut une phase d'analyse évaluant la sécurité du programme. Actuellement, ces analyses sont réalisées par revues de code, simulations et expérimentations réelles. Ces analyses présentent des limitations : elles sont peu ou pas automatisées, coûteuses et elles se basent sur l'intuition d'un humain pour déterminer l'ensemble des entrées qui pourraient mettre en évidence une vulnérabilité.

1.1 Contributions

Cette thèse s'intéresse à l'analyse de robustesse d'un code binaire en présence de fautes intentionnelles. Le parti pris sur le choix du niveau binaire se justifie par la présence, à ce niveau, d'informations invisibles à des niveaux de code plus haut. En effet, l'encodage des instructions ainsi que l'implantation des instructions et des données sont autant

d'informations qui peuvent être sources de vulnérabilités.

L'objectif de cette thèse est de concevoir une méthode et un outil capable d'analyser la robustesse d'un code binaire, de manière automatique, face à des injections de fautes, en utilisant de la vérification formelle, afin d'aider les experts en sécurité ou concepteurs de logiciels sécurisés à déployer les contre-mesures logicielles. La méthode proposée permet la modélisation de fautes transitoires affectant aussi bien les instructions que les données.

Les contributions de cette thèse sont :

— Une méthode d'analyse de robustesse combinant différents types d'analyses de codes ainsi que de la vérification formelle. La méthode de vérification choisie est basée sur de la vérification de modèle borné (*bounded model-checking*). Des analyses statiques et symboliques sont utilisées pour récupérer des informations sur le code (comme les bornes des boucles nécessaires à la modélisation des problèmes de façon bornée) et d'obtenir des informations sur le contexte d'exécution du code à analyser afin de réaliser une analyse vis-à-vis d'états du programme réalistes et ainsi de limiter les faux positifs. L'ensemble du code et les effets des fautes sont modélisés de sorte que l'analyse puisse les exprimer comme des problèmes SMT.

— Un outil nommé `RobustB` implémentant la méthode proposée. `RobustB` automatise la méthode depuis le code source jusqu'à l'obtention des vulnérabilités : il ne requiert aucune connaissance des méthodes et langages formels de la part de l'utilisateur. Il supporte différents types de fautes affectant le code ou les données. Il est combiné avec trois outils externe : `angr` pour l'analyse du binaire, `capstone` pour le désassemblage et un solveur SMT.

— Trois métriques permettant de synthétiser les résultats de l'analyse. Ces métriques sont basées sur la probabilité d'exécution des différents chemins d'exécution du programme. La première donne la sensibilité de chaque instruction, la seconde et la troisième sont des métriques globales, elles indiquent la surface d'attaque moyenne par exécution et la densité moyenne des vulnérabilités par exécution. Elles ont pour but d'aider le concepteur de contre-mesures à cibler les instructions les plus sensibles, ou à comparer différentes versions protégées d'un même code.

L'outil et ses métriques permettent ainsi de vérifier l'efficacité des contre-mesures insérées au niveau du code source, à la compilation, ou directement dans le code assembleur. Il permet l'analyse de la robustesse du code après compilation et ainsi de s'assurer de l'efficacité des contre-mesures après les passes d'optimisations de codes pouvant altérer les contre-mesures. Les métriques permettent la comparaison de différentes versions protégées, et ainsi de rechercher des compromis de sécurité vis-à-vis des performances ou d'adapter des contre-mesures.

1.2 Organisation du manuscrit

La suite de cette thèse se décompose en 5 chapitres.

Dans le chapitre 2, nous présentons les différents moyens disponibles pour réaliser des attaques en fautes, les effets possibles des attaques en faute et la manière dont elles peuvent être exploitées par un attaquant. Nous présentons ensuite les différents moyens de protection des codes et nous nous intéressons en particulier aux contre-mesures logicielles. Nous montrons les avantages et inconvénients du niveau de l'implémentation des contre-mesures logicielles. Nous présentons ensuite le processus de sécurisation d'un code, et notamment l'étape consistant à analyser la robustesse du code. Nous concluons sur les défauts des analyses de robustesse actuellement employées.

Dans le troisième chapitre, nous discutons des différentes méthodes d'analyse de robustesse à travers deux principaux critères : la modélisation de l'effet des fautes et les méthodes d'analyses de comportement. Nous présentons ensuite les différentes méthodes d'analyse de comportement d'un code que nous avons regroupées en deux catégories : méthodes d'analyse concrètes (*e.g.* simulation, exécution native) et symboliques (*e.g.* model-checking, exécution symbolique, résolution SAT/SMT). Ensuite, nous présentons l'état de l'art, où les approches sont organisées par niveau de code analysé : code source, représentation intermédiaire, code assembleur ou binaire et enfin, microarchitecture.

Dans le quatrième chapitre, nous présentons l'approche développée dans cette thèse ainsi que son implémentation dans l'outil `RobustB`. Nous présentons d'abord les étapes permettant l'obtention des informations nécessaires à la modélisation des différentes analyses sous forme de problèmes SMT puis nous présentons la formalisation de ces problèmes et l'interprétation de leurs résultats. Le reste du chapitre est dédié à la présentation de `RobustB`, l'outil qui a été développé tout au long de l'élaboration de l'approche. Nous présentons les différentes étapes de `RobustB` puis nous détaillons ses modules et leurs interactions.

Le cinquième chapitre est consacré aux expériences réalisées avec `RobustB`. Elles sont organisées en six parties où les quatre premières parties proposent de montrer différents usages de `RobustB` et les deux restantes montre la capacité de l'outil à rendre compte de spécificités présentes dans certains codes.

Le sixième et dernier chapitre présente la conclusion et les perspectives de cette thèse.

Chapitre

2 *Contexte et motivations*

Sommaire

Dans ce chapitre, nous donnons le contexte et les motivations des travaux réalisés dans cette thèse. Dans la section 2.1, sont présentées les attaques physiques et plus particulièrement les attaques en faute au travers des différents moyens utilisés pour leur réalisation. Dans la section 2.2, nous présentons comment il est possible d'exploiter les effets des fautes ainsi que les représentations possibles de leurs effets à différents niveaux du matériel et du logiciel. La section 2.3 présente les techniques disponibles pour la protection du code soumis aux attaques en fautes. Dans la section 2.4, nous présentons le processus de sécurisation de code ainsi que différentes techniques pour évaluer la robustesse d'un code. La section 2.5 clôt ce chapitre en résumant les différents points abordés et présente les objectifs de cette thèse.

2.1 Attaques physiques

Exploitées pour la première fois en 1997 par Boneh *et al.* [BDML97], les attaques physiques constituent une classe d'attaques qui a la particularité de nécessiter un accès physique au système visé. Cette proximité offre de nouveaux vecteurs d'interaction avec le système. Lumière, champs électromagnétiques ou température sont autant de moyens par lesquels une attaque physique peut être réalisée.

Les attaques en faute nécessitent de l'attaquant qu'il perturbe le système par un biais physique dans le but de le faire dévier de son comportement nominal. Typiquement, l'attaque modifie le code ou change la valeur de données manipulées par le code. Les modifications induites sur le code ou les données permettent ainsi de récupérer la clé de chiffrement d'un algorithme cryptographique [DMM+13], de réaliser une escalade de privilèges [TM17] ou de contourner des mécanismes de sécurité [VWWM11].

Nous présentons, dans cette partie, différents moyens par lesquels une attaque en faute peut être réalisée, comment ces attaques peuvent être exploitées et enfin les représentations des effets des fautes à différents niveaux d'abstraction du matériel et du logiciel.

2.1.1 Attaques en fautes

La réalisation d'une attaque physique nécessite que l'attaquant applique un stress physique sur le matériel de façon à le forcer à opérer au-delà de ses spécifications. Nous présentons, dans cette partie, différents moyens d'injection de faute ainsi que les effets physiques qui sont à l'œuvre.

2.1.1.1 Rayonnement ionisant

On parle de rayonnement ionisant lorsqu'un rayonnement (des photons) est capable de détacher des électrons de la matière qu'il traverse. La capacité d'un rayonnement à être ionisant dépend du niveau d'énergie des particules ou ondes qui le constituent et du pouvoir d'arrêt de la matière vers laquelle il est dirigé. Les semi-conducteurs, constituant quasi intégralement les circuits intégrés modernes, sont de plus en plus sensibles aux rayonnements du fait de la finesse de gravure croissante. L'ionisation des circuits *Complementary Metal Oxide Semi-conductor* (CMOS) en silicium, constituant la grande majorité des circuits en circulation, se matérialise par la création artificielle d'un canal de conduction entre les deux substrats (N et P) composant le circuit. La présence de ce canal induit un courant qui se propage à travers les blocs logiques jusqu'à l'entrée d'un registre, ce qui peut modifier la valeur de ce registre si le courant est maintenu lors d'un front montant d'horloge.

Depuis le milieu des années 60, le laser est utilisé pour simuler l'effet des rayonnements ionisants d'origine naturelle ou au moins non intentionnels (rayonnements cosmiques ou simplement issu d'éléments radioactifs) sur les semi-conducteurs. Ce n'est qu'en 2002, les avancées technologiques ayant permis la miniaturisation des puces, que Skorobogatov *et al.* contrôlent la modification de n'importe quel bit de données dans une mémoire Static Random-Access Memory (SRAM) à l'aide d'une source lumineuse focalisée [SA02].

Depuis, ce type d'attaque s'est révélé comme faisant partie des plus efficaces tant sur le plan de la précision que sur celui de la reproductibilité. Néanmoins, son application pratique est encore coûteuse et nécessite que le composant soit décapsulé pour que le rayonnement atteigne le silicium.

Dans [RDT13], les auteurs utilisent un laser pour attaquer une implémentation matérielle d'un AES dans un cryptosystème. Ils réalisent des inversions de bits durant l'exécution du programme. Les auteurs parviennent à récupérer la clé de chiffrement utilisée par une analyse différentielle entre le chiffré fauté et du chiffré correct.

2.1.1.2 Augmentation temporaire de la fréquence d'horloge

L'horloge d'un circuit (ou d'une partie d'un circuit) définit la fréquence à laquelle les registres du circuit échantillonnent le signal qui se trouve en entrée. Une fréquence d'horloge maximisant la fréquence de fonctionnement d'un circuit est dimensionnée en fonction du chemin critique, correspondant au chemin ayant le temps de traversée des portes le plus long entre deux éléments mémorisant.

La fréquence maximale de fonctionnement d'un circuit, ou aussi fréquence maximale d'horloge, est la fréquence au-delà de laquelle le temps de propagation du signal entre deux éléments mémorisant devient supérieur à la période de l'horloge. Au-delà de cette fréquence, au moins un registre du circuit échantillonne un signal présentant une erreur à chaque période d'horloge.

Les attaques par augmentation de la fréquence d'horloge, ou *clock glitch*, visent à obtenir cet effet de façon temporaire. On parle alors d'une faute de violation de contraintes temporelles ou plus communément faute de *timing*. Ce type d'attaque a une bonne précision temporelle mais la localisation spatiale de la faute est faible car ses conséquences, qui sont fonctions du délai appliqué à l'horloge, sont entièrement déterminées par le circuit. Malgré cela, Yuce *et al.* ont réalisé en 2016 [YGS+16] une attaque par clock glitch, affectant plusieurs instructions, en tirant parti de la longueur du temps de propagation dans l'étage d'exécution du pipeline du processeur SPARC.

La réduction temporaire de la tension d'alimentation est un autre moyen physique produisant un effet similaire. Réduire la tension d'alimentation ralentit la propagation des signaux à travers la logique du circuit. Si le temps de propagation d'un signal entre deux registres est supérieur à la période entre deux échantillonnages, le registre censé recevoir ce signal échantillonnera un signal présentant une erreur.

2.1.1.3 Induction électromagnétique

L'utilisation de champ électromagnétique, dans le but de perturber le comportement d'un circuit, a pour la première fois été proposée par Quisquater *et al.* en 2002 [QS02]. Dans ce papier, les auteurs induisent un courant à la surface d'un cryptoprocesseur d'une carte à puce en faisant varier le champ magnétique par l'impulsion d'une tension aux bornes d'une bobine placée au-dessus du circuit et dont le noyau est axé perpendiculairement à celui-ci. La variation du champ magnétique engendre, à travers une induction électromagnétique, un courant de Foucault à la surface du circuit qui, à son tour, modifie l'information contenue dans une cellule mémoire.

Les attaques par variation du champ magnétique ont une meilleure précision spatiale que les attaques par clock glitch mais celle-ci est néanmoins plus faible que celle des attaques par rayonnements ionisants. En effet, la variation du champ magnétique, et donc l'induction de courant à la surface du circuit, sont maximales dans l'axe de la bobine mais, bien que faiblissant en s'en éloignant, elles restent tout de même non négligeables [Gho18] et peuvent donc impacter des éléments environnant.

Les variations de champ magnétique affectent les mailles et réseaux du circuit. Un circuit est constitué de plusieurs réseaux (horloge, alimentation). Étant donné la faible localité d'une attaque par impulsion électromagnétique, M. Ghodrati émet l'hypothèse dans sa thèse [Gho18] que les fautes sont le résultat de l'influence combinée de la variation du champ magnétique sur ces différents réseaux. Comme Zussa *et al.* dans [ZDT+14], il infère des résultats d'une faute que l'induction électromagnétique induit une violation de contrainte temporelle, soit par une augmentation temporaire de la fréquence d'horloge, soit par une baisse temporaire du voltage dans le circuit induisant un temps de propagation plus long des portes logiques.

Un second type d'attaques par impulsions électromagnétiques a été proposé. Ces attaques sont plus étendues dans le temps que les précédentes et le champs magnétique varie selon une fonction sinusoïdale. De telles perturbations ont été utilisées par Poucheret *et al.* en 2011 pour perturber le comportement de l'horloge d'un System On Chip (SoC) [PTL+11].

Les attaques à base d'induction électromagnétique offrent moins de contrôle spatial qu'une attaque au laser mais à l'avantage d'être moins coûteux et de pouvoir s'effectuer sans décapsuler le système visé.

2.1.1.4 Rowhammer

Des attaques en faute peuvent aussi être réalisées de manière purement logicielle. C'est le cas de l'attaque *rowhammer*, ou martèlement de mémoire en français, qui exploite un phénomène parasite de certaines mémoires lorsque celles-ci sont sollicitées intensément. Ce phénomène permet de modifier le contenu de certaines cellules d'une mémoire DRAM [KDK+14].

Les mémoires DRAM sont constituées de cellules organisées en lignes et colonnes, représentées respectivement sur l'axe Y et X de la figure 2.1. Chaque cellule est constituée d'une capacité, dont l'absence ou la présence de charge définit la valeur du bit, et d'un transistor commandant l'accès à la donnée (voir figure 2.2). Lors d'un accès en lecture à un mot, la ligne correspondante est mise à l'état haut et les capacités associées à chaque bit de ce mot transfèrent leurs charges au *row-buffer* (nommé amplificateur de détection en français). La charge stockée dans la capacité n'étant pas naturellement persistante, les constructeurs de mémoires DRAM garantissent un temps de rétention de la donnée pendant un laps de temps avant qu'un mécanisme de rafraîchissement vienne rétablir la quantité de charge.

En 2014, Kim *et al.* montrent qu'avec la miniaturisation des mémoires DRAM, rappro-

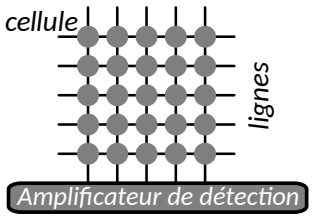

FIGURE 2.1: (source [KDK+14]) Organisation des cellules d'une mémoire DRAM

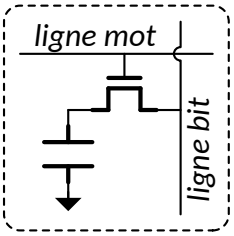

FIGURE 2.2: (source [KDK+14]) Accès à une cellule d'une mémoire DRAM

chant les cellules mémoires les unes des autres et réduisant la quantité de bruit qu'elles peuvent supporter, elles sont plus susceptibles d'interagir électriquement entre elles. Ils montrent cette interaction en accédant de façon répétée et fréquente à la mémoire [KDK+14].

En accédant de façon répétée et fréquente aux lignes adjacentes $n-1$ et $n+1$, une attaque rowhammer induit une corruption de données à la ligne n. Kim *et al.* attribuent ce phénomène à l'accélération de la fuite des charges dans la ligne n qui est telle que la fréquence de rafraîchissement des cellules ne permet plus de conserver la valeur de la cellule.

En 2016, Gruss *et al.* [GMM16] ont réalisé une attaque rowhammer au travers d'un programme écrit en JavaScript. Le programme infère la hiérarchie mémoire de la machine sur lequel il s'exécute pour implémenter une stratégie rapide d'éviction de lignes de cache de façon à déclencher l'effet rowhammer sur la mémoire principale. Les auteurs exploitent cet effet pour prendre le contrôle de la machine.

2.1.1.5 CLKscrew

Les mécanismes de gestion de consommation, visant à optimiser la consommation et étendre la durée de vie des batteries [TSS17], sont maintenant omniprésents dans les systèmes embarqués. L'optimisation de la consommation d'énergie de ces systèmes fait aussi bien intervenir les concepteurs du matériel que du logiciel. Cette coordination se matérialise par la conception de matériels génériques permettant un contrôle dynamique de la tension et, ou de la fréquence d'horloge du système. Pour s'adapter aux besoins changeant du système, les variations de la tension ou de la fréquence d'horloge sont sous le contrôle du noyau au travers de registres adressables.

L'attaque CLKscrew [TSS17], présentée par Tang *et al.* en 2017, est une exploitation par le logiciel de mécanismes matériels dédiés à la gestion de la consommation d'un système, et ce de façon à forcer le processeur du système à opérer au-delà de ses spécifications. Les auteurs illustrent l'attaque en récupérant la clé de chiffrement utilisée par un AES s'exécutant

à un niveau de privilège plus élevé que le noyau (comme le Trusted Environnement Execution (TEE) [1]). Ils synchronisent l'augmentation de la fréquence par le biais d'une attaque par canal auxiliaire (SCA) pour corrompre un octet de la matrice de l'AES à un moment stratégique de l'algorithme AES. Répétant l'opération, ils récupèrent 3500 chiffrés fautés permettant ensuite de déduire la clé avec une analyse différentielle.

2.2 Exploitation et modèles de faute

On a vu précédemment quelques moyens de réalisation d'attaques en faute. L'effet d'une injection de faute dépend de la cible et du logiciel en train de s'exécuter. On appelle *modèle de faute* une représentation de l'effet d'une injection à un niveau donné de représentation du code (binaire, assembleur, source) ou du matériel (logique, micro architectural).

Pour réussir une attaque, un attaquant peut chercher à caractériser les fautes réalisables pour obtenir une représentation des effets à des niveaux du code qui l'intéressent. Cette modélisation des fautes est aussi essentielle à la sécurisation du code : il est nécessaire de déterminer les effets possibles d'une injection sur le logiciel pour s'en prémunir.

On s'intéresse, dans cette partie, aux effets des fautes d'un point de vue attaquant et d'un point de vue concepteur de système.

2.2.1 Exploitation

Du point de vue de l'attaquant, une attaque en faute se décompose en deux phases : une phase d'exploration et une phase d'exploitation. Dans la phase d'exploitation, l'attaquant cherche à déterminer quelles fautes il est capable d'induire sur la cible avec les moyens d'injections à sa disposition. Avant que l'attaquant ne puisse observer les conséquences de ses attaques au niveau logiciel, le moyen d'injection doit être finement réglé. Le nombre de paramètres à régler dépend bien sûr du moyen d'injection utilisé. Par exemple, cinq paramètres doivent être ajustés pour une attaque au laser : la position selon les axes X, Y et Z du laser ainsi que la puissance de l'impulsion et sa durée. Un autre paramètre doit être ajusté lorsque l'impulsion est commandée automatiquement par la détection d'un événement sur la cible, le délai entre la détection de l'événement et l'impulsion. Lorsque l'attaquant utilise un moyen d'injection précis en termes de localité de l'injection, il doit aussi déterminer sur quel élément de la cible il souhaite réaliser l'injection (processeur, mémoire, bus, ...)

1. Trusted Execution Environnement : zone sécurisée du processeur et isolée des autres environnements d'exécution.

puis explorer les différents paramètres de son moyen d'injection (positionnement, puissance, durée, ...). Enfin, l'attaquant doit déterminer pour quel code et pour quel usage de ce code il souhaite exploiter l'attaque (vérification sensible, chiffrement, copie de tampon, ...). Dans les travaux de recherches visant la compréhension des effets des injections, la phase d'exploration peut être réalisée bien plus finement, le travail d'exploration est alors nommé caractérisation [MDH+13, Dur16, CMD+18].

La figure 2.3 montre différents niveaux matériels et logiciels auxquels les effets d'une faute peuvent être représentés. Au niveau physique, la perturbation induite par le moyen d'injection induit à son tour un courant à la surface du circuit. L'intensité de la perturbation physique doit être contrôlée par l'attaquant. Si la perturbation est trop intense la cible pourrait être définitivement endommagée. À l'opposé, si la perturbation est trop faible le fonctionnement de la cible n'est pas perturbé. Au niveau circuit, le courant induit est échantillonné par une ou plusieurs bascules. Les bits contenus dans les bascules fautées sont interprétés par la micro architecture, ils peuvent alors représenter des données, du code ou encore des bits servant à la bonne exécution du code. Le nombre de bits fautés est corrélé avec l'intensité de la perturbation. La faute se propage au niveau Instruction Set Architecture (ISA) lorsque les instructions qui sont exécutées ou les données qu'elles manipulent diffèrent de ce qu'elles auraient dû être en l'absence de faute. Les effets de la faute peuvent alors être observés en sortie du programme et dépendent entièrement du code attaqué. Il peut s'agir de chiffré fauté, d'un *buffer overflow*, ou encore d'une modification du flot de contrôle. L'effet de la faute peut alors être exploité par l'attaquant pour obtenir les bénéfices souhaités.

Timmers *et al.*, dans [TM17], réalisent une escalade de privilège grâce à un *voltage glitch*. Les auteurs injectent la faute pendant l'exécution de l'appel système `setresuid`. Cet appel système permet, entre autre, à un programme de choisir son identifiant. Normalement, l'Operating System (OS) interdit à un programme utilisateur de changer son identifiant en '0', identifiant du super utilisateur *root*. En initialisant les registres à '0' avant l'appel à `setresuid`, Timmers *et al.* parviennent à lancer un programme avec les droits de super utilisateur depuis un code utilisateur.

Dehbaoui *et al.*, dans [DMM+13], attaquent une implémentation d'AES sur un micro-contrôleur ARM Cortex-M3. Les auteurs parviennent à incrémenter le nombre de tours de boucle réalisés par l'algorithme en empêchant l'incrémentation de la variable d'induction de la boucle. Les auteurs récupèrent alors un chiffré fauté qu'ils comparent, par une analyse différentielle, avec le chiffré correct pour récupérer la clé de chiffrement.

Wounderberg *et al.*, dans [VWWM11], contournent une double vérification de code PIN sur une carte à puce. Ils ont au préalable cartographié la sensibilité de la puce puis,

après avoir déterminé le début de l'exécution du code de vérification en analysant la consommation, ils sont parvenus à réaliser deux injections de faute pour contourner les vérifications successives.

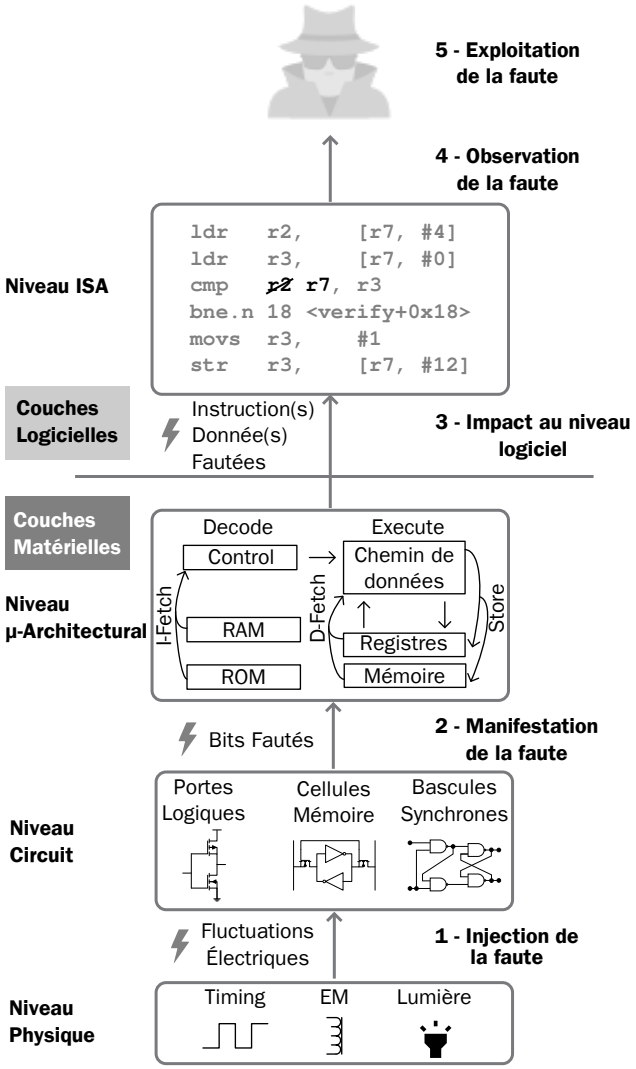

FIGURE 2.3: (source [YSW18]) Propagation et effets d'une faute à différents niveaux de représentation

2.2.2 Modèle de faute

Les modèles de faute sont utilisés autant par les attaquants que par les concepteurs de contre-mesures. L'attaquant, lorsqu'il connaît le code qu'il souhaite attaquer, peut avoir besoin d'avoir une abstraction des fautes qu'il peut réaliser au niveau de représentation du code qu'il a à sa disposition. Le concepteur de contre-mesures doit pouvoir se représenter l'effet des fautes au niveau où il opère. Nous abordons ici la représentation des fautes qu'il est possible de construire aux niveaux logiciels et matériels ainsi que les différents niveaux de persistance d'une faute.

2.2.2.1 Persistance d'une faute

Un modèle de faute est caractérisé par une représentation des effets d'une faute et par la persistance de la faute dans le temps. On distingue trois persistances [BECN+06] : les fautes destructives, transitoires et persistantes.

Les fautes destructives sont la conséquence d'un dommage matériel sur le circuit comme l'altération de l'intégrité d'une cellule mémoire ou d'un registre. L'élément affecté peut alors avoir un comportement dit de collage (ou *stuck-at*) où la valeur d'un bit reste inchangée une fois la faute injectée [YSW18].

Les fautes transitoires n'ont pas d'effet permanent sur le circuit, c'est le cas des attaques que nous avons déjà décrites : un courant est induit par une perturbation physique, l'état du signal associé au courant est échantillonné par une bascule produisant la faute. L'état de la bascule est réinitialisé lors de la prochaine écriture ou par la réinitialisation du circuit. Les fautes transitoires sont les plus recherchées par les attaquants car elles leur permettent de réaliser plusieurs fautes, sans risque de destruction du circuit, et ce jusqu'à l'obtention de l'effet voulu.

Les fautes persistantes sont un cas particulier des fautes transitoires. Ces fautes persistent durant toute la durée du programme, comme par exemple, la modification d'une instruction dans un binaire chargé en RAM. L'instruction ou la donnée fautée, si elle se trouve dans une boucle, affecte l'exécution du programme à tous les tours de la boucle. Les fautes persistantes incluent aussi le cas où la faute modifie une donnée en RAM et où celle-ci n'est pas écrite pendant l'exécution du programme.

2.2.2.2 Niveau circuit

On représente les fautes au niveau circuit par un effet sur la valeur d'une bascule. Cet effet peut être soit la mise à '1' (*bit-set*), la mise '0' (*bit-reset*) ou l'inversion de la valeur de la bascule (*bit-flip*). Le nombre de bascules affectées peut varier selon le moyen d'injection utilisé et l'intensité de l'injection. Le laser est généralement utilisé pour affecter un ou plusieurs bits contigus tandis que les émissions électromagnétiques ont généralement un champ d'action plus étendu et moins précis (plusieurs bits).

2.2.2.3 Niveau jeu d'instructions (ISA)

La modélisation des fautes au niveau du jeu d'instructions dépend du rôle des bascules affectées dans le circuit. Les modèles de fautes les plus fréquemment observés à ce niveau sont les suivants :

— Saut d'instruction (*instruction skip*) : une instruction n'est pas exécutée. C'est un des modèles de faute les plus fréquemment mis en évidence [BGV11, KMW17]. Par exemple, il peut être obtenu par la modification du codage binaire de l'instruction de sorte que l'instruction modifiée n'ait plus d'effet sur le reste du programme [MDH+13].

— Modification d'un bit d'une instruction (*instruction bit set/reset/flip*) : le codage de l'instruction change, par la modification d'un de ses bits, et le mot fauté représente une instruction valide [CMD+18].

— Corruption de registre (*register corruption*) : la valeur d'un registre est remplacée par une autre. L'effet de ce modèle de faute peut-être atteint de diverses manières : modification d'un ou de plusieurs bits (bit flip/set/reset) ou encore, modification d'une donnée transitant depuis la mémoire vers le processeur. Dans ce modèle de faute, le registre destination de l'instruction visée prend une autre valeur à l'issu de son exécution [BOS03, VKS11b].

— Remplacement d'instruction : une autre instruction est exécutée à la place de l'instruction initialement attendue. Ce modèle englobe la modification d'un bit dans le codage d'une instruction et le saut d'instruction. En plus de ces deux modèles, il inclut aussi le cas de la modification de plusieurs bits d'une instruction [BGV11, BBKN12, MDH+13], qui peut être atteint, par exemple, par une corruption à l'étage *fetch*, étage chargé de récupérer une instruction depuis la mémoire (cache ou RAM).

Des modèles de fautes liés à la microarchitecture ont aussi été observés. Rivière *et al.* rapportent des expériences où une injection électromagnétique sur un processeur Cortex-M4 induit la répétition de quatre instructions [RNR+15]. Ils en déduisent que l'injection électromagnétique annule le remplissage du tampon de préchargement du processeur (*prefetch buffer*) laissant ainsi son contenu inchangé.

2.2.2.4 Niveau source et algorithmique

Au niveau source ou algorithmique, les instructions assembleur et leurs détails d'exécution sont inconnus et ne peuvent donc pas être pris en compte. Tous les modèles de fautes observés à plus bas niveau n'ont donc pas forcément d'équivalent au niveau source. Les effets les plus communément représentés au niveau du code source sont : la corruption de

variable, la modification du flot de contrôle ou encore la modification d'une instruction d'un programme source.

Bouffard *et al.*, au travers d'une injection laser, montrent qu'il est possible de modifier le flot de contrôle d'un code s'exécutant sur une Javacard en modifiant l'adresse de retour d'une fonction [BICL11].

Berthomé *et al.*, dans [BHKTL12], transcrivent les effets de fautes au niveau assembleur vers le langage C. Les fautes qu'ils transcrivent impactent uniquement le flot de contrôle. Les auteurs se basent sur deux modèles de fautes au niveau assembleur : un nombre arbitraire de sauts d'instructions contiguës et la modification du codage d'une instruction en une instruction de saut. Le premier modèle est équivalent à sauter un nombre arbitraire d'instructions "en avant" tandis que le second est équivalent à sauter un nombre arbitraire d'instructions "en avant" et "en arrière". Les auteurs représentent ces deux modèles de fautes au niveau C par des instructions `goto`. L'exécution des instructions `goto` peuvent alors être conditionnées pour représenter une faute transitoire.

2.3 Contre-mesures

Face à la menace que constituent les attaques par perturbation, plusieurs contre-mesures logicielles et matérielles ont été proposées dans le but d'améliorer la sécurité des circuits et des codes.

Que ce soit au niveau matériel ou logiciel, les contre-mesures doivent être conçues vis-à-vis d'un modèle de faute spécifiant l'effet de la perturbation au niveau où elles sont conçues [VKS11a]. Qu'elles soient matérielles ou logicielles, les contre-mesures peuvent être distinguées selon leurs objectifs : tolérer la faute ou la détecter. Dans le premier cas, la contre-mesure vise à insensibiliser le circuit ou le code à une perturbation donnée, autrement dit, en présence ou non de cette perturbation, le comportement macroscopique du système doit rester le même. Dans le second cas, la contre-mesure vise à détecter l'effet de la perturbation puis à déclencher un mécanisme adéquat correspondant aux attentes de confidentialité/intégrité des informations manipulées par le système (destruction, réinitialisation ...). Il est aussi possible d'infecter le calcul de façon à le rendre inutilisable par l'attaquant sans lui divulguer que l'attaque a été détectée.

Nous présentons dans cette partie les principales contre-mesures matérielles et logicielles ainsi que les processus de conception et déploiement de ces dernières.

2.3.1 Contre-mesures matérielles

Des contre-mesures matérielles de détection et de tolérance ont été proposées au niveau circuit. Elles ont pour but de protéger les circuits contre les effets des perturbations ou leur source elles-mêmes. Les contre-mesures visant la tolérance peuvent prendre la forme de mécanismes de compensation ou même de bouclier visant l'imperméabilité du circuit à une source de perturbation. Les contre-mesures de détection comprennent l'ajout de capteurs pour détecter l'effet d'une perturbation ou la source de la perturbation.

El-baze *et al.* [EBRM16] ainsi que Zussa *et al.* [ZDT+14] ont proposé deux contre-mesures au niveau circuit impliquant l'intégration de capteurs détectant les effets d'une injection électromagnétique à ce niveau, i.e. une faute de timing. Borel *et al.* [BDD+18] ont proposé une contre-mesure au niveau physique par l'ajout d'un bouclier conducteur sur la face arrière protégeant la puce contre le décapsulage, nécessaire à la réalisation d'une attaque au laser. Les capteurs embarqués visent à prémunir le système contre des sources de perturbations ou effets d'une source de perturbation bien définie et ne permettent donc pas, à eux seuls, une protection totale du circuit.

Des contre-mesures basées sur de la redondance temporelle ou spatiale, afin d'effectuer plusieurs fois les mêmes calculs, sont souvent utilisées. La redondance spatiale implique la multiplication de blocs matériels et la vérification de la cohérence des données en sortie des blocs. Il s'agit alors de détection de faute lorsque les blocs sont dupliqués, mais il est possible de réaliser de la tolérance aux fautes en triplant le bloc à protéger [BECN+06]. La valeur correcte est alors décidée par vote majoritaire. Cette technique a l'avantage de ne pas être restreinte à une source de perturbation mais reste tout de même sensible aux sources de perturbation ayant une action affectant entièrement le circuit.

Qu'elles soient basées sur de la redondance ou sur des capteurs, les contre-mesures matérielles constituent un élément important dans la panoplie disponible pour la protection de systèmes. Néanmoins elles souffrent de certains inconvénients. L'ajout de matériel coûte cher, tant en termes de conception que de déploiement, et les protections matérielles ne sont pas flexibles. En effet, l'efficacité des protections décroît naturellement au fur et à mesure que de nouvelles attaques, ou moyens d'attaque, sont découverts. Suite à la découverte d'une nouvelle attaque, les contraintes de sécurité peuvent ne plus être respectées. Un nouveau système incluant de nouvelles protections matérielles doit être repensé, produit et déployé en remplacement des anciens. Un tel processus coûte cher, ce qui n'est pas le cas des contre-mesures logicielles.

2.3.2 Contre-mesures logicielles

Comme les contre-mesures matérielles, les contre-mesures logicielles se conçoivent à tous les niveaux de représentations du code. Il existe plusieurs types de contre-mesures logicielles, celles qui nous intéressent ici se présentent sous la forme d'ajout de code autour de la fonctionnalité à protéger.

Dureuil *et al.* ont proposé, par exemple, un ensemble de codes protégés au niveau source [DPP+16]. Cet ensemble de codes se veut représentatif des protections génériques (*e.g.* intégrité du flot de contrôle) qui peuvent être implémentées pour des applications sensibles. Les contre-mesures déployées font entrer en jeu des duplications de variables et des vérifications de la cohérence des variables, finales ou intermédiaires, avec leur duplicata.

Lalande *et al.* ont proposé, en 2014, une contre-mesure au niveau source détectant, dans une fonction écrite en C, un saut en avant ou en arrière d'au minimum deux instructions source. La protection consiste en l'ajout de compteurs d'étapes (*steps counters*) et d'un test de la valeur du compteur entre chaque instruction du code source garantissant ainsi l'intégrité du flot de contrôle séquentiel lors de l'exécution du programme. Des variables supplémentaires permettent de vérifier l'intégrité du flot de contrôle.

Akkar *et al.* ont proposé, en 2003, un outil de transformation de code associé à un ensemble de directives préprocesseur pour la gestion automatisée de structures visant la détection de fautes perturbant le flot de contrôle [AGL+03]. Cet outil et ces directives ont pour but de simplifier la mise en place de schémas de protection basés sur des steps counters.

Les contre-mesures insérées au niveau source sont très attrayantes pour leur portabilité et la simplicité de leur déploiement comparativement au niveau assembleur. Néanmoins, elles sont soumises à la phase de compilation et notamment aux phases d'optimisation de celle-ci. Ces optimisations visent, entre autres, l'élimination de code redondant et l'élimination de code ne produisant pas d'effet de bord, deux propriétés inhérentes à un grand nombre de contre-mesures insérées au niveau source. Pour prévenir la suppression des contre-mesures, il est souvent nécessaire de recourir à des techniques pour empêcher ponctuellement (*volatile* en C), partiellement (compilation en `O0` de la fonction sensible) ou complètement, les optimisations du compilateur de s'appliquer.

Dans sa thèse, Nicolas Moro, propose un schéma de tolérance au modèle de faute du saut d'une instruction assembleur [Mor14]. Le schéma de tolérance se déploie au niveau

assembleur. Il consiste en la duplication des instructions idempotentes [2] du code. Les instructions n'ayant pas cette propriété dans le code original sont transformées en une suite d'instructions idempotentes équivalentes avant d'être dupliquées. Les instructions du code dupliquées permettent alors au code d'être robuste contre un saut d'instruction. Cette approche a l'avantage de ne pas soumettre la contre-mesure à l'épreuve des passes d'optimisation du compilateur mais double, au moins, le temps d'exécution et la taille du code généré. En effet, en plus de la duplication des instructions, la transformation des instructions non idempotentes en une séquence d'instructions idempotentes équivalentes introduit des instructions supplémentaires et nécessite l'utilisation de plus de registres. Ce schéma peut introduire de nouvelles lectures et écritures mémoire pour stocker le contenu des registres afin de les libérer. Dans [BCR16], les auteurs automatisent l'application de ce schéma de protection en l'intégrant dans un compilateur et améliorent ainsi le coût en performance et en taille du code final par la spécialisation de la sélection d'instructions et par l'allocateur de registres.

L'insertion automatique de protections lors de la compilation a aussi été étudiée dans la thèse de Julien Proy [Pro19]. J. Proy a intégré au compilateur, après les passes d'optimisations du cœur du compilateur (*middle-end*), l'insertion automatique d'un schéma de protection garantissant le bon nombre d'itérations dans une boucle contre un saut d'instruction ou une corruption de registre. Ces travaux montrent l'intérêt de se placer dans le compilateur pour déployer des protections. En effet, protéger le code au sein du compilateur permet de contourner les principaux inconvénients de l'insertion de protections au niveau du code source (déployées à ce niveau, elles peuvent être mises en défaut par le compilateur) et au niveau binaire (perte de sémantique). Toutefois, la modification du compilateur pour l'insertion automatique de protection n'est pas un problème simple : il ne s'agit pas seulement de placer habilement la passe de sécurisation dans le cœur du processus de compilation mais il est aussi nécessaire de modifier ou désactiver des passes dans le back-end (allocation de registres) pour éviter qu'elles n'altèrent ou éliminent les protections.

En plus des problèmes déjà soulevés concernant l'insertion de contre-mesures logicielles, un autre problème, commun à toutes, est le fait que l'ajout de code, associé à l'insertion de protections logicielles, augmente automatiquement la surface d'attaque. Louis Dureuil a appelé ce problème le "paradoxe de la surface d'attaque" [Dur16]. L'ajout de contre-mesures dans un code augmente nécessairement le nombre de points où il est possible d'injecter une faute (points d'injection), le rende potentiellement plus vulnérable.

Ainsi, quelle que soit la méthode et le niveau choisi pour l'insertion de protections

2. Instruction idempotente : instruction pour laquelle le résultat de son exécution est indépendant du nombre successif de fois où elle est exécutée.

logicielles, la sécurité du code binaire final n'est pas garantie : les protections peuvent être mal conçues, mal déployées ou altérées par le compilateur. De plus, comme mentionné ci-dessus, l'ajout de protections logicielles augmente nécessairement la surface d'attaque du code sur lesquelles elles sont déployées. Il est donc important d'analyser la sécurité du code final.

2.4 Processus de sécurisation

Le processus de sécurisation comprend trois étapes : la conception de la contre-mesure, son déploiement et l'analyse du code protégé. La première phase consiste en la conception d'une technique de protection dédiée à un code ou à un modèle de faute. Dans l'étape de déploiement, la protection est implémentée dans le code cible (au même niveau que celui où elle a été conçue). Dans l'étape de vérification, la sécurité du code final est évaluée en utilisant diverses techniques. Ce processus suit souvent un schéma itératif d'essais-erreurs jusqu'à l'obtention du niveau de sécurité voulu, *i.e.* jusqu'à ce que le résultat de la phase de vérification soit satisfaisant et qu'un bon compromis performance/sécurité soit trouvé.

L'analyse du code protégé constituant la phase de vérification peut prendre plusieurs noms, on parle d'analyse de robustesse ou de recherche de vulnérabilités. Trois approches différentes d'analyse de robustesse du logiciel sont souvent utilisées. La première, la revue de code, consiste en la lecture du code assembleur par un expert. Outre l'aspect laborieux de la lecture du code assembleur, déduire "à la main", et de façon exhaustive, les conséquences d'une faute à ce niveau est difficile, voir irréalisable sur des codes trop complexes. La seconde approche est le test de pénétration où les fautes sont réellement injectées sur le système complet. Cette approche nécessite des équipements spécifiques et, plus restrictif encore, un haut niveau d'expertise concernant l'équipement nécessaire à l'injection de faute. Ces deux approches sont utiles mais restent néanmoins limitées en termes de couverture des fautes possibles. Une autre possibilité est l'approche par simulation de faute où la faute peut être simulée à travers l'utilisation d'un débogueur [PHBC17] ou directement dans le code [LHB14].

Ces trois approches ne peuvent être exhaustives. D'un côté le cerveau humain n'est pas capable d'appréhender toutes les conséquences d'une injection de faute pour toutes les variables d'entrée et toutes les valeurs fautées possibles. De l'autre, l'exécution directe du code ou l'utilisation de techniques de simulation impliquent que les paramètres d'entrée du programme soient fixés. L'ensemble des comportements possibles du programme face à une faute ne peut donc pas toujours être simulé : par exemple, déterminer la robustesse d'un code à une corruption de registre nécessite de considérer toutes les valeurs possibles de

tous les registres utilisés tout au long du code et sur tous les chemins possibles. Certaines vulnérabilités sont susceptibles de n'apparaître qu'avec une configuration spécifique des entrées et des valeurs issues de fautes. Les combinaisons possibles des valeurs des entrées et des valeurs issues d'une faute sont souvent bien trop nombreuses pour être totalement explorées par ces techniques.

Du fait du coût des solutions actuelles, de leurs limitations et du besoin de vérification de code, il y a un réel besoin de solutions automatisées permettant d'étendre la portée des techniques existantes. Ce besoin est discuté dans la prochaine section.

2.5 Besoin de vérification automatisée de la robustesse de code binaire

Nous avons montré dans ce premier chapitre que les systèmes embarqués sont de plus en plus susceptibles d'être la cible d'attaques physiques et notamment d'attaques en faute. Ces attaques permettent, à travers la manipulation de quantités physiques dans l'environnement du système, de contourner des mécanismes de sécurité ou encore de réaliser une escalade de privilèges.

Les contre-mesures conçues pour protéger les systèmes contre cette classe d'attaques sont imparfaites. En plus d'augmenter la surface d'attaque, l'insertion de contre-mesures logicielles est particulièrement fragile face aux passes d'optimisations du compilateur. La perte de sémantique inhérente aux représentations de code bas niveau impacte la précision des protections qui peuvent être déployées à ce niveau et en conséquence, la performance du code final.

Actuellement, les approches existantes visant à évaluer la sécurité d'un code, revues de code, simulation ou tests de pénétration, manquent d'exhaustivité et, mise à part la simulation, elles sont peu ou pas automatisées.

Il y a donc un réel besoin de méthodes et d'outils pour analyser la robustesse du code binaire final. Réaliser une analyse au niveau binaire permet de prendre en compte le codage des instructions assembleur, le placement de code, les effets des optimisations du compilateur ainsi que des modèles de fautes plus proches de la réalité.

Les travaux menés dans cette thèse ont pour but de répondre à ce besoin en proposant une approche et un outil visant la satisfaction de trois critères. Le premier critère est la réalisation de l'analyse de robustesse à bas niveau, qui permet de se placer après le

compilateur et d'analyser le code qui sera effectivement exécuté. Le deuxième est l'utilisation d'une méthode permettant de ne pas se reposer sur l'intuition d'un humain pour déterminer les configurations des paramètres d'entrée potentiellement dangereuses. Le troisième et dernier critère est l'automatisation, très prisée dans le cadre d'un développement industriel, qui permet une analyse rapide et à moindre coût.

Le prochain chapitre présente les approches existantes pour l'évaluation de code soumis à des attaques en faute de façon à pouvoir situer l'approche retenue dans cette thèse.

Chapitre

3 État de l'art sur l'analyse de robustesse contre les attaques en faute

Sommaire

Ce chapitre présente tout d'abord une définition de l'analyse de robustesse. Vient ensuite une présentation des différentes façons de représenter une faute et l'impact de chacune de ces représentations sur le résultat d'une analyse de robustesse. Suit une présentation des méthodes utilisées pour analyser le comportement d'un code ou d'un système en présence de fautes. Enfin, nous présentons les approches existantes pour l'analyse de robustesse en mettant l'accent sur la représentation des fautes et la méthode d'analyse utilisée avant d'en faire une synthèse.

3.1 Introduction à l'analyse de robustesse

Un programme robuste est un programme qui conserve son comportement général dans un environnement perturbé (en présence de faute) ou qui est capable de détecter la perturbation. Ce qui correspond au comportement général doit être défini a priori ; cette définition peut prendre la forme d'un ensemble de propriétés fonctionnelles à satisfaire ou spécifie le comportement du programme dans un environnement non perturbé. L'environnement perturbé définit le modèle de menace, et dépend des capacités de l'attaquant et de la criticité du composant à analyser : type de faute, temps disponible à l'attaquant pour réaliser les attaques.

3.1.1 Analyse de robustesse

Les outils et méthodes d'analyse de robustesse contre les attaques en faute cherchent à déterminer si un programme soumis à une faute est robuste (ou dans quelle proportion il n'est pas robuste) en fonction d'un modèle de menace. Pour cela, ils représentent une faute sur le code (ou dans le système), analysent le comportement du code (ou du système) et cherchent à garantir deux types de propriétés :

— La détection : conservation d'une partie pertinente des comportements nominaux ou détection de la faute. Vise à garantir que, soit la faute n'a pas d'effet sur ce qui doit être conservé, soit elle est détectée.

— La tolérance : conservation de l'intégralité des comportements nominaux.

Pour réaliser l'analyse, les comportements qui doivent être conservés doivent être connus de l'utilisateur et, dans certains cas, ils sont transmis à l'analyse de robustesse. L'information transmise à l'analyse, qui encode ces comportements est appelée propriété de sécurité. Le comportement du code fauté est ensuite analysé et les fautes qui invalident la propriété de sécurité sont découvertes.

3.1.2 Précision sur les modèles de menace

Tous les produits ne doivent pas répondre à un même niveau de sécurité. Le niveau de sécurité d'un produit peut être déterminé par son usage, plus ou moins sensible. Il peut par exemple s'agir de dispositifs de communication militaire ou de téléphones portables. L'environnement, public ou privé, dans lequel le produit évolue est aussi pris en compte pour déterminer un niveau de sécurité satisfaisant. Le niveau de sécurité qu'un produit doit garantir détermine les caractéristiques des attaquants potentiels (leurs niveaux d'expertise et leurs moyens) dont découlent les modèles de fautes qu'ils peuvent injecter. Les Critères Communs [Cri] [1] déterminent sept niveaux de sécurité (EAL [2]) indiquant un niveau d'évaluation des vulnérabilités (classe AVA). Plus le niveau de sécurité d'un produit est élevé, plus l'analyse des vulnérabilités d'un produit est réalisée en profondeur, et plus les attaquants considérés sont experts et ont à leur disposition des moyens (temps et équipements) importants.

3.2 Analyse de comportement en présence de faute

Dans cette partie, nous présentons différentes modélisations de l'effet des fautes ainsi que les différentes méthodes utilisées par les outils d'analyse de robustesse pour analyser le comportement du code ou du système en présence de fautes.

3.2.1 Modélisation des effets des fautes

Comme présenté dans la partie 2.2.1, les effets des fautes peuvent être modélisés à tous les niveaux, depuis les portes logiques jusqu'au code source. Des analyses de robustesse ont été développées à chacun de ces niveaux et ont ainsi eu besoin de modéliser les effets des fautes au niveau qu'elles considèrent. Dans le chapitre précédent (cf. section 2.2.2), nous avons présenté les différents niveaux de modélisation des fautes ainsi que les limites de chacun de ces niveaux. On a notamment vu que les effets des fautes observés à bas niveau ne sont pas toujours transposables à haut niveau. Pour les mêmes raisons, une analyse faite à haut niveau ne peut pas représenter tous les effets des fautes observables à plus bas niveau.

1. Critères Communs : ensemble de normes internationales destinées à l'évaluation de la sécurité de systèmes et de logiciels. Ces évaluations ont pour but de déterminer le niveau de sécurité du produit et ainsi délivrer la certification associée.
2. EAL : Evaluation Assurance Level

On peut classer les techniques de modélisation des effets des fautes en deux catégories : statiques ou dynamiques. Les techniques statiques représentent l'effet des fautes avant l'évaluation du système (*e.g.* modification de la valeur d'un bit d'une instruction dans un code binaire) générant ainsi des codes dits mutants. De telles modélisations ne permettent pas, telles quelles, de simuler des fautes transitoires. Par exemple, l'effet d'une faute ciblant une instruction se trouvant au milieu d'une boucle peut impacter l'évaluation du code à chaque tour de boucle. Pour contourner ce problème, il est possible d'ajouter du code conditionnant l'apparition de la faute (*e.g.* pour déclencher l'effet de la faute à une itération en particulier). De tels ajouts de codes nécessitent toutefois de s'assurer que leurs présences n'ajoutent ni ne suppriment des vulnérabilités qui auraient été autrement trouvées.

Les techniques dynamiques, quant à elles, représentent l'effet des fautes au cours de l'évaluation du système. En simulation, l'effet de la faute peut être appliqué à un ou plusieurs éléments lors de la simulation (*e.g.* incrémentation du compteur ordinal pour simuler un saut d'instruction). Les analyses qui exécutent le code directement sur du matériel peuvent obtenir les mêmes effets via des mécanismes logiciels comme des interruptions, des services du système d'exploitation ou encore en utilisant les fonctionnalités d'un debugger. Pour la modélisation de l'effet de fautes transitoires, les techniques dynamiques sont en général préférables aux techniques statiques car elles n'ont pas de problèmes d'interférences mais requièrent tout de même un moyen d'altérer le système lors de son évaluation.

3.2.2 Méthodes d'analyse de comportement

On distingue deux grandes familles : les analyses se basant sur des méthodes concrètes et celles se basant sur des méthodes symboliques. Ces deux termes, concret et symbolique, portent sur la représentation des configurations d'entrée ainsi que sur la représentation des états d'un système que l'analyse peut considérer. En effet, les méthodes dites concrètes instancient les valeurs de tous les éléments (*e.g.* variables, registres) en entrée du code analysé. Elles permettent de décrire un comportement de façon la plus précise possible compte tenu du niveau de représentation considéré. Mais, à cause du grand nombre de combinaisons de valeurs possibles, il leur est difficile de couvrir l'ensemble des comportements d'un programme et offrent donc peu de garanties vis-à-vis de la robustesse d'un programme. Au contraire, les méthodes d'analyse du comportement dites symboliques peuvent considérer des valeurs abstraites (c'est-à-dire les valeurs possibles dans un ensemble des valeurs concrètes) en entrée du code analysé et sont donc en mesure de couvrir tous les comportements possibles d'un programme. Pour chaque variable abstraite, l'ensemble de définition associé peut être un sous-ensemble, l'ensemble exact ou une sur-approximation du domaine des valeurs concrètes de la variable. Dans le cas d'une sur-approximation, il est

possible d'introduire des comportements qui ne correspondent pas à la réalité et mènent alors à des faux positifs. Dans le cas d'une recherche de vulnérabilité il est alors possible de trouver des vulnérabilités qui ne peuvent pas être réalisées en conditions réelles.

Dans les prochaines parties, nous présentons les techniques basées sur ces deux familles de méthodes qui sont classiquement utilisées dans l'analyse de comportement de programme.

3.2.2.1 Analyses concrètes

Les analyses de robustesse concrètes comprennent les méthodes d'analyse du comportement d'un programme basées sur de l'exécution native ou de la simulation. Ces deux techniques nécessitent de fixer toutes les valeurs d'entrée du programme analysé. Réaliser une analyse pour toutes les combinaisons possibles de ces valeurs n'est pas possible dans le cas général. Les méthodes concrètes nécessitent souvent de l'utilisateur qu'il fixe l'ensemble des valeurs des entrées à analyser et limite l'espace de recherche à celles-ci. En revanche, ces méthodes n'imposent pas, ou peu, de contraintes sur la taille du code à analyser.

L'exécution native. Le comportement du programme est directement produit par l'exécution de celui-ci sur une plateforme d'exécution matérielle. Dans le cadre d'une analyse de robustesse, les méthodes exécutant le code nativement doivent trouver un moyen de simuler l'effet de la faute. Ainsi, dans [BBC+14], les auteurs ont ajouté un service dans le système d'exploitation d'une carte à puce de façon à pouvoir modifier l'état du système lors de son exécution. Ils reproduisent ainsi les effets de fautes par altération du flot de contrôle, saut d'instruction ou encore corruption de registre. Proy *et al.*, dans [PHBC17], exécutent le programme à travers un debugger et utilisent les fonctionnalités (points d'arrêt, contrôle du contenu des registres) de ce dernier pour simuler les effets de fautes de type saut d'instruction et corruption de registre.

La simulation [3]**.** Elle permet d'analyser le comportement du code en abstrayant les couches matérielles et logicielles du support d'exécution. Ce support d'exécution est tout de même modélisée par le simulateur qui est chargé du maintien de sa cohérence tout au long de la simulation. La simulation peut s'effectuer à différents niveaux d'abstraction : porte logique, microarchitecture, assembleur/binaire ou encore au niveau du code source. Les effets des fautes représentables dépendent du niveau de la modélisation sous-jacente à la simulation. L'avantage de la simulation par rapport à une exécution native est le contrôle qu'elle apporte à l'injection de faute. Par exemple, une modélisation au niveau des portes logiques a permis à Faurax de modifier le délai de portes logiques [Fau08] générant

3. Simulation : le mot simulation peut aussi être employé pour caractériser la simulation de la faute injectée et non l'évaluation du code. Nous parlons ici de simulation du code.

des fautes de type bit-flip. Un niveau de modélisation plus proche du matériel augmente le nombre d'éléments à modéliser ainsi que le nombre d'éléments sur lesquels peuvent porter les fautes, mais cela augmente aussi la durée de la simulation. L'effet des fautes peut aussi être directement représenté sur le code en modélisant celui-ci avant de l'exécuter sur un simulateur [BHKTL12] ou du matériel.

3.2.2.2 Analyses symboliques

La seconde famille de méthodes correspond à celles opérant symboliquement. Ces méthodes visent à considérer tous les comportements possibles du code ou du système, *i.e.* elles sont exhaustives (notamment lorsque les domaines symboliques sont des sur-approximation des domaines concrets). Dans le cadre d'une analyse de robustesse contre une attaque en faute, cela permet de déterminer si un code est robuste à une faute, *i.e.* elles peuvent montrer qu'il n'existe pas de configurations des valeurs des entrées du système pour lesquelles la faute constitue une vulnérabilité. Toutefois ces méthodes sont en général sujettes à l'explosion combinatoire ce qui limite la taille des problèmes solvables et leur complexité, ou elles approximent fortement le problème (comme c'est le cas pour l'interprétation abstraite qui est le plus souvent utilisée pour prouver la correction de programme). Les méthodes symboliques peuvent générer des faux positifs (*i.e.* indiquer qu'un programme est vulnérable alors que l'analyse porte sur une trace qui n'a pas d'équivalent concret), et nécessitent donc souvent que les résultats des analyses soient investigués plus en profondeur lorsqu'ils indiquent la découverte d'une vulnérabilité.

Les méthodes formelles sont depuis longtemps utilisées pour vérifier la conformité de logiciels ou de matériels vis-à-vis de leurs spécifications fonctionnelles. Elles sont également utilisées dans le cadre de la démonstration automatique de théorèmes [Sch02] et la vérification de démonstrations [BC13] ; dans ce second cas, on parle d'assistant de preuve. Ces méthodes ont pour point commun de proposer des cadres de raisonnement partiellement ou totalement automatisés.

Dans le cadre de la vérification de matériel, les méthodes formelles ont permis une exploration des comportements possibles du matériel plus efficace que la simulation ou les méthodes énumératives qui étaient alors utilisées. Elles ont notamment permis le développement d'algorithmes symboliques d'analyse des comportements (*i.e.* manipulant des ensembles d'états et non plus les états concrets un à un). Dans cette lignée, les ***BDD*** (*Binary Decision Diagram* pour Diagramme de Décision Binaire), introduits sous sa forme actuelle par Bryant en 1986 [Bry86], sont des diagrammes permettant de représenter la fonction caractéristique d'ensembles représentant des états booléens et des transitions

entre états booléens sous forme de graphes orientés acycliques où chaque chemin, de la racine à une feuille, décrit une affectation des variables de la formule.

Une autre approche, utilisée avec succès pour la vérification de logiciels et de matériels au tournant des années 2000 et qui s'est amplifiée depuis, est la résolution d'un problème de satisfiabilité booléenne, communément appelé *SAT*. Dans cette approche, les problèmes à résoudre (dont on cherche l'existence d'une solution) sont décrits sous la forme de logique propositionnelle incluant des conjonctions ($\bigwedge$), des disjonctions ($\bigvee$), des négations ($\neg$) et des variables propositionnelles. La résolution d'un problème SAT revient à déterminer l'existence ou la non-existence de configurations des variables d'une formule, constituée de variables booléennes et de connecteurs logiques, pour laquelle celle-ci est vraie.

Model-checking. Appelée vérification par modèle en français, cette technique de vérification formelle consiste à déterminer si les comportements d'un système décrits sous la forme d'un graphe des états accessibles, est un modèle pour une propriété donnée. L'ensemble des propriétés forme la spécification comportementale du système qui peut être exprimée dans un formalisme logique. En général, une logique temporelle LTL [4] ou CTL [5] est utilisée, car elles présentent des algorithmes de vérification relativement efficace comparée à d'autres logiques plus expressives.

Une des approches les plus utilisées dans le cadre de la vérification de programmes est le *Bounded Model-Checking* ou *BMC*. Cette technique permet de résoudre des problèmes trop grands pour être résolus avec des techniques de model-checking classiques en restreignant le problème à des séquences finies. Même si cette méthode est incomplète par essence, elle reste utilisée pour la vérification de matériel et de logiciel car elle permet tout de même d'analyser une partie significative des comportements d'un système. En 2000, Sheeran *et al.*, ont proposé la *k-induction* [SSS00], une approche basée sur la méthode SAT et du model-checking qui permet la complétude de la résolution. Cette méthode nécessite l'explicitation d'invariants, ce qui requiert, dans la plupart des cas, une intervention humaine et n'est pas applicable à tous les systèmes.

Satisfiability Modulo Theories ou *SMT*, est une technique de résolution SAT enrichie par des théories étendant l'expressivité des problèmes analysés. Les théories (*e.g.* sur les réels, entiers ou vecteurs de bits) introduisent un ensemble de fonctions et de prédicats permettant une expression plus naturelle des problèmes. Les théories introduisent aussi

4. Linear Temporal Logic (LTL) : logique temporelle linéaire en français, LTL est une logique permettant de décrire des propriétés sur l'ordre d'occurrence d'événements le long de séquences d'états. Elle est basée sur les connecteurs de la logique propositionnelle avec, en plus, des opérateurs temporels.

5. Computational Tree Logic (CTL) : contrairement à LTL, CTL permet d'exprimer des propriétés considérant l'ensemble des chemins possibles et introduit des quantificateurs de chemin. Toutefois, CTL n'est pas capable d'exprimer certaines propriétés exprimables en LTL (notamment les propriétés de stabilité)

des procédures de décision qui sont utilisées par le solveur pour interpréter les éléments de la formule. En pratique, les solveurs SMT utilisent deux solveurs pour résoudre une formule modulo une théorie : un solveur SAT et un solveur de la théorie. La première étape est la transformation de la formule SMT en une formule SAT en attribuant une variable booléenne différente à chaque prédicat. Le problème SAT est ensuite donné au solveur SAT interne et celui-ci renvoie **insatisfiable** s'il ne trouve pas d'affectation des variables booléennes pour lesquelles la formule est vraie. Sinon le solveur de la théorie est appelé pour vérifier la cohérence de l'affectation des variables booléennes dans le domaine de la théorie. Si le solveur de la théorie ne trouve pas d'incohérence, le solveur SMT renvoie **satisfiable**. Sinon il ajoute une nouvelle clause au problème SAT qui explicite l'incohérence trouvée par le solveur de théorie et la recherche recommence à la première étape [DMB11].

Une autre approche visant elle aussi à contourner le problème d'explosion combinatoire du model-checking classique est le *Statistical Model-Checking* ou la vérification statistique de modèle en français. Cette technique combine de la simulation avec des méthodes statistiques et cherche à déterminer la probabilité que le système simulé satisfasse une propriété. Elle se base sur des échantillons représentatifs de résultats de simulation et, avec des méthodes statistiques, permet de déterminer si la probabilité qu'un système satisfasse une propriété est plus grande ou plus petite qu'un seuil donné. Même si cette méthode ne permet d'obtenir qu'une garantie statistique, elle permet tout de même de borner la probabilité de l'erreur. Cette méthode peut être théoriquement utilisée pour résoudre des problèmes trop importants ou trop complexes pour être résolus avec du model-checking classique tout en apportant une garantie statistique par rapport à une simulation classique [LDB10]. Toutefois, la garantie statistique est entièrement dépendante du modèle probabiliste qui a permis la génération de l'ensemble d'échantillons représentatifs. De plus, l'élaboration d'un tel modèle est un problème compliqué.

Exécution symbolique. Contrairement à la simulation qui nécessite des valeurs pour toutes les variables d'entrée du programme simulé, l'exécution symbolique, introduite au milieu des années 70 [Cla76], représente les variables de façon symbolique. L'exécution des instructions du programme ajoute de nouvelles relations (ou contraintes) entre les variables symbolisées. Cette méthode utilise un solveur SMT pour déterminer la validité des expressions obtenues le long des différents chemins d'exécution. Cette vérification est réalisée lors de l'exécution d'un branchement conditionnel. Le moteur d'exécution symbolique maintient en permanence une représentation de l'ensemble des contraintes rencontrées le long d'un chemin d'exécution et lorsqu'un branchement doit être exécuté, il fait appel à un solveur SMT pour déterminer la faisabilité des deux chemins résultants, limitant ainsi l'exécution symbolique aux chemins d'exécution faisables [God11].

Cette approche passe difficilement à l'échelle à cause de l'explosion combinatoire potentielle du nombre de chemins et/ou des états mémoire induits par les accès mémoires dépendant de variables symboliques. En effet, le moteur d'exécution symbolique considère tous les chemins de façon concurrente, ce qui résulte en la croissance exponentielle du nombre de chemins en fonction du nombre de branchements conditionnels rencontrés. De plus, une adresse symbolique peut faire référence à n'importe quelle case de la mémoire, ce qui multiplie le nombre d'états à prendre en compte par le moteur d'exécution [BCD+18].

Pour résoudre ce problème, une méthode alternative d'exécution symbolique a été développée et est très utilisée, pour la couverture de code, dans le cadre de la recherche de bugs : il s'agit de l'**exécution symbolique concrète** dite aussi concolique [WMM04]. Contrairement à l'exécution symbolique pure, la version concolique peut concrétiser certaines variables, *i.e.* leur attribuer une valeur au cours de la simulation. Le but de la concrétisation est de diminuer la complexité du problème dans le but de pouvoir y répondre. La valeur d'une variable peut, par exemple, être concrétisée lorsque l'expression de celle-ci devient trop complexe et donc plus difficilement solvable par le solveur SMT. L'analyse ne peut alors plus rendre compte de l'ensemble des comportements du programme : tant en ce qui concerne la couverture de code que des configurations, l'analyse n'est plus exhaustive. L'exhaustivité de l'analyse est sacrifiée au profit de sa performance (sans quoi celle-ci aurait été trop longue ou même dans l'incapacité de répondre). Toutefois, des heuristiques existent et permettent de limiter la perte de l'exhaustivité engendrée par la concrétisation. Il reste tout de même difficile de garantir que tous les états pertinents ont été explorés. La concrétisation et alors utile lorsque l'objectif est de trouver des vulnérabilités mais elle perd en pertinence lorsque l'objectif est de garantir la robustesse.

3.3 Outils d'analyse de codes contre les attaques en faute

Dans cette partie, nous nous intéressons aux outils d'analyse de robustesse contre les attaques en faute proposés dans la littérature. Historiquement, les outils d'analyse visant l'évaluation de la robustesse d'un code face à des fautes ont été développés dans le cadre des fautes non intentionnelles. C'est seulement à partir de 1997 [BDML97], date de la première attaque par perturbation, que la question du développement de méthodes et d'outils pour modéliser ce phénomène s'est posée.

Il existe des différences notables entre les fautes non intentionnelles et celles intentionnelles qui nous amènent à la séparation de ces deux mondes. Dans le cas des fautes non

intentionnelles, il est possible de déterminer les menaces actuelles et futures (essentiellement environnementales), auxquelles le système sera soumis, au moment de sa conception. De plus, l'espace d'apparition des fautes dans le système est beaucoup plus large dans le cadre des fautes non intentionnelles. Ceci est principalement dû au fait que, lors d'une attaque par perturbation (faute intentionnelle), l'attaquant cherche à casser le système d'une façon qui lui permet de l'exploiter. À l'inverse, une faute non intentionnelle peut arriver n'importe où même si, en pratique, des composants sont plus sensibles que d'autres. Les systèmes dans lesquels des protections contre les fautes non intentionnelles doivent être mises en place sont des systèmes critiques, pour lesquels la sûreté est d'une importance capitale (par exemple dans le domaine de l'aérospatial). La tolérance aux fautes non intentionnelles doit, de façon analogue aux pannes matérielles, être garantie sur l'intégralité du système pour assurer son bon fonctionnement en toute circonstance.

Dans cette partie, nous discriminons les différentes approches et outils en fonction du niveau de modélisation de l'effet des fautes analysées. Nous présentons les approches existantes modélisant les fautes du niveau code source jusqu'au niveau micro-architectural en mettant l'accent sur les avantages et inconvénients liés au niveau et à la méthode choisis pour analyser le comportement du programme.

3.3.1 Analyse de robustesse au niveau du code source

Les outils d'analyse de robustesse travaillant sur des modélisations des fautes au niveau source, ou même algorithmique, ont l'avantage de détecter les vulnérabilités au plus tôt dans le processus de développement. Ces analyses sont portables car indépendantes du matériel sur lequel les codes analysés sont déployés.

En 2007, **Larsson *et al.*** ont été les premiers à proposer d'utiliser l'exécution symbolique pour analyser le comportement du code [LH07] en présence de fautes non intentionnelles. Leur méthode considère du code source Java et analyse l'impact de fautes transitoires de type bit-flip dans les variables globales du programme. Ils ont pu prouver qu'une implémentation de l'algorithme *CRC* (Cycling Redundancy Check), réalisant la correction d'erreurs lors d'une transmission de données, détecte bien toutes les fautes de type bit-flip réalisées sur les données transmises.

Berthomé *et al.* [BHKTL10, BHKTL12, LHB14] ont proposé d'analyser les effets de fautes ayant pour conséquences un saut en avant ou en arrière dans un code source. Les auteurs ont observé que les effets de plusieurs modèles de faute à plus bas niveau peuvent se modéliser comme un saut au niveau du code source. Leur approche simule les sauts

possibles par la création de codes mutants résultant de l'insertion d'une instruction C `goto` dans le code source. L'exécution des instructions `goto` est conditionnée de façon à simuler l'effet de fautes transitoires. Les mutants sont ensuite compilés puis exécutés. Le résultat d'une exécution d'un code mutant est ensuite rangé parmi six classes : *bad* (la faute peut constituer une vulnérabilité, le mutant doit être investigué), *good* (le résultat est conforme à l'exécution non fautée, la faute n'a pas eu d'effet), *not triggered* (la faute n'a pas été injectée par non-exécution du code contenant la simulation de l'effet de la faute) et les trois cas restant constituent des erreurs à l'exécution. Cette approche supporte un unique modèle de faute, le saut en avant ou en arrière intraprocédural dans le code C. Les résultats de l'analyse sont présentés sous la forme d'un graphique 2D montrant le résultat de tous les sauts possibles d'une instruction source (axe X) à une autre (axe Y) au sein d'une même fonction. Les auteurs se sont servis de cette analyse pour montrer la robustesse d'une fonction C à tous les sauts d'une distance de deux instructions C ou plus.

Rauzy *et al.* [RG14] ont proposé une méthode et un outil, `finja`, pour analyser formellement des versions de l'algorithme cryptographique *CRT-RSA* (*Chinese Rest Theorem RSA*) contre des attaques en faute. L'outil opère sur une représentation algorithmique, transformée en un arbre, et réalise des simplifications sur cet arbre. Les fautes sont simulées en changeant la propriété (valeur retournée) des nœuds. Les modèles de faute supportés sont la mise à zéro et la corruption du résultat d'une opération. La propriété que l'algorithme doit conserver pour qu'il soit considéré robuste à la faute simulée est exprimée en fonction des termes représentés dans l'arbre. Rauzy *et al.* se sont servis de cette méthode pour prouver la robustesse d'une implémentation de l'algorithme CRT-RSA incluant une contre-mesure [ABF+02] contre l'attaque BellCore [BDML97].

3.3.2 Analyse de robustesse au niveau d'une représentation intermédiaire

Les analyses réalisées au niveau d'une représentation intermédiaire [6] (IR) gardent l'avantage de la portabilité du code source tout en analysant une représentation du code plus proche de celle finalement exécutée. Les modèles de faute représentables au niveau IR sont ainsi plus proches de ceux observés à bas niveau qu'au niveau source, mais tous les modèles de faute observés à bas niveau ne peuvent pas être représentés au niveau IR [PHBC17]. L'analyse à ce niveau permet, là encore, la détection de vulnérabilités au plus tôt sans toutefois garantir la robustesse du code final.

6. Représentation intermédiaire : représentation du code d'un programme se situant à mi-chemin entre le code source et le code assembleur. Une telle représentation est notamment utilisée par les compilateurs pour faciliter la manipulation du code dans certaines passes d'optimisation ne nécessitant pas de prendre en compte une architecture en particulier.

Machemie *et al.*, ont proposé un outil de simulation réalisant l'injection de fautes persistantes dans du *bytecode* pour Java Card [MMLC11]. Cet outil génère des codes mutants en modifiant un par un chaque octet du bytecode par une valeur choisie, `0x00` ou `0xff` selon le type de mémoire visé. Un interpréteur abstrait est ensuite chargé d'analyser le comportement de chacun des mutants. Si, pendant cette exécution, aucune erreur n'est détectée ou si une exception est levée et n'est pas capturée, alors la faute est considérée comme non détectée. Tous les mutants pour lesquels la faute n'a pas été détectée sont décompilés et le code Java résultant doit être analysé par l'utilisateur pour déterminer comment la faute s'est propagée dans le code. Les auteurs ont pu trouver de multiples points d'attaque dans des programmes pour Java Card réalisant des opérations critiques (géolocalisation, génération de mots de passe, protocole de paiement).

Potet *et al.* ont proposé `Lazart` [PMPD14], un outil pour l'analyse des vulnérabilités d'un code soumis à de multiples fautes transitoires de type inversion de test de branchement conditionnel. `Lazart` analyse le code au format LLVM IR[7]. Il opère sur le graphe de flot de contrôle du programme (*CFG*) qu'il analyse. Le CFG d'un programme encode tous les chemins d'exécution possibles : il est composé de blocs de base contenant les séquences atomiques d'instructions du programme et d'arcs représentant les transitions possibles, par saut ou en séquence, entre les blocs de base. À partir d'un objectif d'attaque exprimé comme l'accessibilité ou la non-accessibilité d'un bloc de base, `Lazart` détermine s'il existe une faute ou une combinaison de fautes permettant la réalisation de cet objectif. L'outil utilise une technique de coloriage de graphe pour déterminer les blocs à partir desquels l'exécution est susceptible d'atteindre le bloc cible et ceux à partir desquels cela n'est plus possible. À partir de cette information, `Lazart` génère un code mutant comportant toutes les exécutions issues de toutes les inversions de tests conditionnels possibles visant l'atteinte (ou non) du bloc objectif. Le code est ensuite exécuté par *KLEE* [CDE+08], un moteur d'exécution concolique au niveau LLVM IR. Il y a trois résultats possibles à l'issue de l'exécution concolique du code : (1) *le code est robuste* : toutes les combinaisons ont été explorées et aucune d'entre elles ne constitue une vulnérabilité, (2) *l'analyse n'est pas concluante* : aucune combinaison explorée ne constitue une vulnérabilité mais l'analyse est incomplète[8], (3)*une vulnérabilité a été trouvée* : au moins une combinaison d'inversions de test a permis d'atteindre le bloc objectif. `Lazart` est, à notre connaissance, le seul outil capable de réaliser des fautes multiples de façon exhaustive. Sa principale limitation réside dans l'unique modèle de faute considéré. Les auteurs ont, entre autres, pu trouver deux chemins d'attaque dans l'implémentation de OpenSSH 6.2[9] d'une fonction de détection

7. LLVM IR : représentation intermédiaire du compilateur LLVM
8. Incomplétude : ici, l'incomplétude de l'analyse constitue un cas où le solveur, intégré au moteur d'exécution symbolique, n'a pas pu répondre. Cela peut être dû à une explosion du nombre de chemins ou à une complexité trop grande des contraintes à résoudre.
9. https ://www.openssh.com/

d'une faille du protocole `ssh`.

3.3.3 Analyse de robustesse au niveau assembleur ou binaire

Les analyses au niveau assembleur ou binaire ont accès à des éléments non visibles à des niveaux de représentation plus élevés. Le placement de code, les registres utilisés par les instructions, les instructions et leurs encodages sont autant d'éléments qui n'existent pas au niveau IR et moins encore au niveau du code source. Tous ces éléments peuvent être impactés par l'effet d'une faute et ainsi participer à l'émergence de vulnérabilités qui ne sont pas détectables à un niveau supérieur de représentation du code.

Dans sa thèse, **P. Andouard** a proposé `OSCAR` [And09], un simulateur de microcontrôleur 8-bit Atmel dédié à l'analyse par canal auxiliaire de la consommation de courant induite lors de l'exécution d'un code soumis à une attaque en faute. La faute à réaliser par le simulateur est spécifiée par l'utilisateur et le modèle de faute supporté est la modification d'un registre ou du compteur ordinal par une valeur prédéterminée. `OSCAR` génère des traces de consommation de courant à l'aide d'un modèle et permet ainsi à l'utilisateur d'investiguer la présence d'éventuelles fuites d'informations en comparant les traces de consommation en présence d'une faute avec celles issues d'exécutions non fautées. L'auteur a notamment mis en avant des vulnérabilités dans une implémentation des programmes `GZIP` et `BZIP2`, deux logiciels de compression.

Pattabiraman *et al.* ont proposé `SymPLAID` [PNKI09], un outil réalisant l'évaluation de la robustesse d'un code assembleur MIPS vis-à-vis de corruptions de données, en combinant exécution symbolique et méthodes formelles, basées sur du model-checking. `SymPLAID` est basée sur un autre outil, `SymPLFIED` [PNKI08], proposé par les mêmes auteurs. Ces deux outils opèrent en décrivant la sémantique de chaque instruction d'un programme assembleur MIPS au travers d'équations et de réécriture qui sont interprétés par `Maude` [CDE+03]. `SymPLAID` met l'accent sur des attaques émanant de codes malveillants qui peuvent être contenus dans des bibliothèques. Il permet l'énumération exhaustive de toutes les corruptions de données (registres et emplacements mémoire) en des points prédéfinis du programme avec un ensemble de valeurs d'entrée prédéfinies ainsi que la modification du compteur ordinal à une valeur connue. Les valeurs des paramètres d'entrée du programme ainsi que la propriété à analyser sur les valeurs de sortie doivent être fournies par l'utilisateur. Les auteurs ont pu trouver une vulnérabilité, par corruption du pointeur de pile, dans le programme OpenSSH menant à l'authentification de l'attaquant. `SymPLFIED` prend en compte des attaques pouvant impacter le code n'importe quand lors de son exécution mais ne propage pas de valeurs. En effet, `SymPLFIED` injecte des erreurs

symboliques qui sont propagées dans les registres et la mémoire en tant que symbole le long des différentes exécutions possibles. Le long de ces exécutions, l'outil vérifie ensuite la validité de propriétés fournies par l'utilisateur.

Berthier *et al.* ont proposé `EFS` [BBC+14], une technique pour simuler l'effet d'une faute sur un code exécuté sur carte. En utilisant l'interface *ADPU* (*Application Protocol Data Unit*), l'exécution du code est stoppée à un point du programme puis l'effet de la faute est simulé avant de reprendre l'exécution. Cette technique permet de simuler et d'analyser la robustesse de code en présence d'une faute ayant un effet sur le code (*i.e.* saut d'instruction) ou les données (*i.e.* corruption de registre). Les auteurs parviennent à simuler le modèle de faute de saut d'instructions et mesurent dans le même temps la consommation de la carte dans le but de localiser les vulnérabilités du code. Dans le prolongement de ce travail, **Rivière *et al.*** ont combiné `Lazart` et `EFS` [RPL+14]. Les auteurs ont utilisé `Lazart` pour détecter de potentielles zones sensibles dans le code puis `EFS` pour simuler les effets du saut de plusieurs instructions contigües prenant alors en compte les spécificités de la carte et du jeu d'instructions. La simulation de l'effet de la faute est, comme utilisé dans les travaux de Proy *et al.* [PHBC17], réalisée à travers l'utilisation des fonctionnalités d'un debugger. La combinaison de ces deux outils a permis de réduire d'un facteur 10 le temps d'analyse d'une fonction de vérification de code PIN par rapport à une analyse réalisée avec `Lazart` seul. De plus, cela a permis de trouver davantage de vulnérabilités.

Dans sa thèse, **L. Dureuil** a proposé `CELTIC` [Dur16], un outil destiné à aider les évaluateurs des *CESTI*[10] qui réalisent des campagnes d'attaques par injection de fautes réelles sur des composants dans le but de produire un rapport pour leur certification. `CELTIC` combine un modèle de faute avec les paramètres du moyen d'injection, ainsi qu'avec les dimensions spatiale et temporelle de l'injection (resp. élément mémorisant affecté et moment de l'exécution auquel la faute advient) pour construire un modèle de faute probabiliste. Ces modèles de faute probabilistes doivent être construits à partir d'attaques par perturbation réalisées au préalable sur le composant visé. L'outil évalue le comportement d'un programme par l'exécution de ses instructions par un simulateur, il permet la représentation de multiples fautes transitoires et nécessite de l'utilisateur qu'il indique la valeur initiale de tous les éléments mémorisants pertinents à la simulation d'un programme donné. Les modèles de fautes probabilistes sont basés sur des modèles de fautes préalablement définis : incrémentation du compteur ordinal, corruption d'une lecture en mémoire volatile par une valeur choisie, corruption d'une lecture de registre par une valeur choisie et écriture d'une valeur choisie dans la mémoire volatile ou dans un registre. Les exécutions résultant d'une injection de faute sont filtrées à l'aide d'un oracle définissant les

10. CESTI : organisme agréé, indépendant, chargé d'évaluer le niveau de sécurité d'un produit selon des normes définies, avant sa mise sur le marché.

conditions de succès d'une attaque pour un programme donné. Louis Dureuil a proposé deux métriques pour évaluer la sécurité globale de l'application et la dangerosité de chaque faute. Il a notamment pu trouver des vulnérabilités liées à l'inversion d'un branchement conditionnel, du remplacement d'une instruction par un saut et de la modification de l'adresse d'un saut dans une fonction d'authentification par code PIN.

Goubet *et al.* ont proposé une méthode basée sur la vérification formelle, implémentée dans un outil pour évaluer la robustesse d'un code assembleur contre les attaques en fautes [GHEDK16]. L'outil, `RobustA`, prend en entrée une description d'un programme ARMv7-M (Thumb-2) sans contre-mesures (A_0) et une autre du même programme contenant des contre-mesures (A_h). Deux analyses sont réalisées : la première vise à déterminer l'équivalence de A_0 et A_h (*i.e.* garantir que les deux programmes produisent le même résultat pour des entrées identiques), la seconde vise à déterminer la robustesse de A_h en présence d'une faute parmi les modèles de faute de saut d'instruction, remplacement d'instruction, et corruption de registre. Pour ses analyses, `RobustA` construit une représentation d'un programme sous la forme d'un automate qu'il déroule de façon bornée. Les fautes sont modélisées dans `RobustA` par ajout ou par modification de transitions dans l'automate déroulé. Les automates déroulés sont ensuite traduits sous forme de problème SMT. La détection de vulnérabilités se fait par *equivalence-checking* de la représentation formelle de A_0 avec celle de A_h. Le problème, donné à un solveur SMT, contient ainsi la représentation des deux automates déroulés et une propriété de sécurité. Cette dernière est dérivée d'une propriété donnée par l'utilisateur, qui définit les éléments qui doivent être équivalents en un point de l'exécution pour que la faute ne soit pas considérée comme une vulnérabilité. Autrement dit, `RobustA` considère que la faute mène à une vulnérabilité si les valeurs d'un même élément dans A_h et A_0 diffèrent à la fin de l'exécution des automates. Les auteurs ont notamment pu prouver la robustesse contre un saut d'instruction d'une contre-mesure [BBK+10] appliquée à une instruction de lecture mémoire. `RobustA` a aussi permis d'étendre cette même contre-mesure au modèle de faute de remplacement d'une instruction. La thèse présentée ici a comme point de départ cette approche.

Proy *et al.*, ont proposé de simuler l'injection de faute sur du code binaire s'exécutant sur une carte ayant un processeur ARM Cortex-M [PHBC17]. L'injection est réalisée à travers l'utilisation d'un debugger. Le code s'exécute sur la carte jusqu'à l'atteinte d'un point d'arrêt où une faute transitoire, un saut d'instruction ou la corruption d'un registre par une valeur choisie, est injectée. Une vulnérabilité est détectée manuellement par la comparaison de la sortie des exécutions fautées avec celles non fautées. Les auteurs ont utilisé cette technique pour analyser la robustesse d'une contre-mesure visant la sécurisation d'une boucle contre un saut d'instruction ou une corruption de registre.

Jafri a proposé, dans sa thèse, une approche construite par assemblage de briques logicielles (certaines existantes) pour la recherche de vulnérabilités dans un binaire x86 à l'aide de méthodes formelles [Jaf19]. Le processus part d'un code source où la condition de victoire d'une attaque est décrite sous forme d'une assertion en C (`assert`). Le code source contenant la propriété est ensuite compilé. Un utilitaire modifie le code binaire pour y introduire une faute et produire des codes binaires mutants. Les fautes injectées sont ainsi persistantes. Les modèles de fautes supportés par cette approche sont l'inversion d'un bit (bit-flip), la modification de la cible d'un saut, la mise à zéro d'un octet, le saut d'instruction et la mise à zéro d'un mot. Les binaires mutants dont l'exécution ne produit pas d'erreur et le binaire original sont traduits en des modèles formels qui sont ensuite donnés à un solveur SMT. La faute présente dans un binaire mutant est considérée comme menant à une vulnérabilité si les résultats de l'`assert` du model-checking d'un binaire mutant diffèrent de celui du binaire original. Jafri a aussi proposé une autre approche pour analyser la robustesse d'un binaire ARMv7-M (Thumb-2). Dans cette approche, la propriété n'est plus représentée sous forme d'assertion, mais elle est directement donnée au solveur SMT. Cela permet d'une part d'éviter les interférences avec le compilateur (changement des optimisations possibles, modification du placement de code, ...) qui limitent les garanties sur les vulnérabilités trouvées. D'autre part, cela évite que leur outil d'injection de faute, opérant au niveau binaire, cible la propriété. Jafri a notamment mis en avant des vulnérabilités dans l'algorithme cryptographique PRESENT [BKL+07] qui permettent de contourner intégralement l'encryptage des données. Les vulnérabilités trouvées dans cet algorithme ont toutes comme source l'altération d'instructions de saut.

3.3.4 Analyse de robustesse au niveau de la micro-architecture

Effectuer l'analyse et représenter les fautes en prenant en compte l'architecture permet à l'utilisateur d'obtenir une compréhension plus fine de la propagation des fautes depuis les éléments architecturaux jusqu'au code.

Jenn *et al.* ont proposé en 1994 l'outil `MEFISTO` qui permet d'injecter des fautes au niveau VHDL [11] [JAR+95] à travers les commandes du simulateur VHDL. Cet outil permet de sélectionner les fautes à réaliser pendant la simulation parmi un ensemble de possibilités très diverses, *e.g.* tous les signaux transportant un type de valeurs ou encore toutes les variables d'un composant. Les auteurs ont pu mettre en avant des vulnérabilités dans plusieurs algorithmes de tri s'exécutant sur une modélisation VHDL du processeur DP32.

11. VHDL : Very high speed integreted circuit Hardware Description Language, est un langage de description matériel, qui, associé à un simulateur, permet la simulation du matériel décrit.

Durant sa thèse, en 2008, **Faurax** a proposé `PAFI` [Fau08], un outil de simulation de faute au niveau des portes logiques d'une implémentation matérielle d'algorithme cryptographique décrite en Verilog [12]. L'outil simule, à travers l'utilisation des commandes du simulateur, l'apparition de délais dans les portes logiques ou de bit-flip dans les mémoires. Les résultats de la simulation fautée peuvent être comparés avec ceux d'une simulation non fautée pour déterminer la présence ou non d'une vulnérabilité. L'auteur a pu mettre en avant des vulnérabilités, liées à l'introduction de délais supplémentaires au niveau des portes logiques, dans plusieurs implémentations matérielles de l'algorithme cryptographique AES.

Schirmeier *et al.* ont proposé, en 2015, `FAIL*` [SHD+15], une infrastructure d'exécution modulaire proposant une analyse très fine des résultats. `FAIL*` est un outil de simulation au niveau architectural. Il est construit autour d'une architecture client serveur où, côté client, l'utilisateur spécifie les paramètres de l'analyse à réaliser : architecture, modèles de faute, valeur des paramètres fixes, valeur des paramètres variables ou encore des évènements particuliers à observer (*e.g.* interruptions). Du côté serveur, `FAIL*` repose sur des simulateurs existants (`GEM5` [BBB+11], `Bochs` [Law96]) et les instrumente pour récupérer des informations sur l'état du système. Les résultats des analyses sont ensuite traités pour fournir à l'utilisateur des informations sur les points sensibles ainsi qu'une aide aux placements de contre-mesures susceptibles d'être efficaces contre les fautes détectées. Les auteurs ont pu évaluer la robustesse d'un système d'exploitation, dont le but principal est de garantir la fiabilité du système aux fautes matérielles. Ils ont pu mettre en évidence des milliers de vulnérabilités liées au modèle de faute bit-flip.

3.3.5 Synthèse des approches à l'analyse de robustesse contre les attaques en faute

Dans cette partie, nous présentons d'abord une synthèse des avantages et inconvénients des éléments principaux avec lesquels nous comparons les différents approches, à savoir : la méthode d'analyse du comportement et la modélisation de l'effet des fautes. Nous présentons ensuite un tableau récapitulatif des approches que nous avons présentées.

12. Verilog : langage de description matériel

3.3.5.1 Synthèses des éléments de comparaison des outils d'analyse de robustesse

Comme nous avons pu le voir, il existe un grand nombre d'approches possibles, combinant différentes méthodes d'analyse du comportement d'un code et à différents niveaux de modélisation de l'effet des fautes qui ont toutes leurs avantages et leurs inconvénients.

Les méthodes d'analyse concrètes. Ces méthodes, regroupant exécution native et simulation, impliquent de fixer les valeurs de tous les éléments mémorisant simulés ou implémentés et nécessitent une nouvelle analyse du code pour chaque configuration des valeurs des entrées pertinentes, fournies par l'utilisateur ou déduites, en amont de l'analyse, par l'utilisation d'une autre technique. Elles ont toutefois l'avantage de leur inconvénient : en limitant la portée de l'analyse à une configuration donnée du système, elles sont capables d'analyser un code en un temps plus court et donc d'augmenter la taille maximum de code analysable en une durée raisonnable. Contrairement à la simulation, l'exécution native a l'avantage de ne pas reposer sur une modélisation du système et n'apporte donc pas d'abstraction dans la représentation du code ou du système. La simulation à plus bas niveau que le code binaire offre toutefois plus de contrôle sur l'injection de faute que dans une exécution native. En effet, dans une exécution native, la simulation de l'effet d'une faute est en général réalisée avec l'aide d'un debugger ou de services de l'OS qui ne peuvent qu'affecter des éléments accessibles par le logiciel (registres, cases mémoire) et en aucun cas des éléments de plus bas niveau comme les registres d'un pipeline ou des portes logiques. Cependant, la simulation d'éléments bas niveau affecte grandement la vitesse de simulation. Celle-ci dépend essentiellement de la précision à laquelle est modélisé le code ou le système : plus la simulation est précise, plus il y a d'éléments à simuler et donc plus la simulation est lente.

Les méthodes d'analyse symboliques. Elles ont l'avantage de ne pas avoir nécessairement besoin de fixer les valeurs initiales des entrées. Elles peuvent ainsi analyser le comportement d'un code ou d'un système en considérant toutes les entrées possibles. Dans le cadre d'une analyse de robustesse contre des attaques en faute, cela permet de rechercher exhaustivement une configuration des valeurs des entrées pour laquelle la faute peut mener à une vulnérabilité. Cette méthode est toutefois sujette aux faux positifs : sans contrainte sur les entrées d'un programme, l'analyse peut renvoyer des vulnérabilités impliquant une configuration des entrées qui ne peut exister lors d'une exécution réelle. Aussi, les fautes transitoires, trivialement réalisées avec les techniques d'analyse concrètes, sont potentiellement plus complexes à représenter dans le cadre d'une analyse symbolique non basée sur des codes mutants. En effet, la modélisation de l'effet de fautes transitoires par une technique dynamique (cf. section 3.2.1) et pour une analyse symbolique nécessite

de l'approche qu'elle ait un moyen d'explorer et d'évaluer les chemins d'exécution existants pour pouvoir y modéliser l'effet des fautes. Un autre aspect limitant est la combinatoire intrinsèque à ces méthodes, qui limite la taille des codes analysables de façon significative en comparaison avec une approche concrète.

Modélisation de l'effet des fautes. Une analyse représentant les fautes au niveau du code source profite d'une grande portabilité ainsi que d'une grande rapidité en comparaison avec une analyse réalisée à plus bas niveau. Toutefois, les modèles de faute qui peuvent être représentés à haut niveau sont limités : le manque de correspondance entre modèles de faute à haut et bas niveau ainsi que les écarts qu'il peut y avoir entre un code source et un code binaire/assembleur ne permettent pas de garantir que toutes les vulnérabilités ont été trouvées. De plus, les passes d'optimisations du compilateur peuvent supprimer du code et ainsi invalider les vulnérabilités qui ont été trouvées à plus haut niveau. Le niveau IR permet de représenter des fautes de façon plus réaliste qu'au niveau source, mais, contrairement au niveau binaire ou assembleur, les représentations intermédiaires sont sujettes à certaines passes du compilateur qui peuvent encore altérer le code [Pro19]. À ce niveau, certaines informations, telles que les registres utilisés ou encore le placement de code, sont inconnues. Les analyses au niveau IR restent néanmoins un bon compromis entre la réalité des fautes et la rapidité de l'analyse tout en conservant une très bonne portabilité. Le niveau binaire donne accès à des informations qui influent sur les conséquences des fautes : registres utilisés par les instructions, adresses d'implantation du code et des données, placement de code. Il permet ainsi de trouver des vulnérabilités qui en découlent. Ce niveau impacte la performance de l'analyse,et la portabilité est réduite aux processeurs partageant le même jeu d'instructions. Une analyse au niveau de la micro-architecture est quant à elle utile pour analyser un système complet et toutes les conséquences des fautes sur des éléments architecturaux. Mais un tel niveau de précision est, dans le cas général, trop cher à simuler.

3.3.5.2 Récapitulatif des approches présentées

Le tableau 3.1 présente l'ensemble des approches précédemment présentées. La première colonne indique le niveau de modélisation des effets des fautes, la seconde colonne contient le nom de l'approche, la troisième rappelle la méthode d'analyse du comportement du système, la quatrième colonne renseigne la persistance des fautes, la cinquième colonne décrit les modèles de faute supportés et la sixième colonne indique la présence de métriques en tant qu'aide à l'analyse des résultats.

TABLE 3.1: Comparaison des outils d'analyse contre des attaques en faute. La colonne **AC** indique la méthode d'analyse du comportement du code : **Sym** pour Symbolique et **Cnc** Concrète. La colonne **FP** indique le modèle de persistance des fautes représentable : **T** pour Transitoire et **P** pour persistante. La colonne **M** indique la présence de métriques

Niveau d'injection de la faute	**Outil**	**AC**	**FP**	**Modèles de faute**	**M**
Algorithmique	`finja` [RG14]	Sym	T	Mise à zéro de variables, corruption de variables	✗
Code source	[BHKTL10] [BHKTL12]	Cnc	T	Saut de minimum 2 instructions C	✗
	[LH07]	Cnc	T	Bit-flip	✗
Représentation intermédiaire	`Lazart` [PMPD14]	Sym	T	Inversions de tests	✓
	[MMLC11]	Cnc	P	Corruption d'un octet du bytecode	✗
Code assembleur et binaire	[PHBC17]	Cnc	T	Corruption de registre, saut d'instruction	✗
	`OSCAR` [And09]	Cnc	T	Corruption de registre, corruption de PC	✗
	`SymPLAID` [PNKI09]	Sym	T	Corruption de registre, corruption de PC	✗
	[BBC+14]	Cnc	T	Corruption de registre, saut d'instruction	✓
	[Jaf19]	Sym	P	Bit-flip/reset, saut d'instruction, corruption de destination de saut, *word-reset*	✗
	`RobustA` [GHEDK16]	Sym	T	Saut d'instruction, corruption de registre, remplacement d'instruction	✗
	`CELTIC` [Dur16]	Cnc	T	Corruption mémoire et registre et leurs lectures, incrémentation de PC	✓
Micro-architectural	`MEFISTO` [JAR+95]	Cnc	T	Corruption de signaux, corruption de variables interne aux composants	✗
	`PAFI` [Fau08]	Cnc	T	Délais porte logique, bit-flip mémoire	✓
	`FAIL*` [SHD+15]	Cnc	T	simple et multiple bit-flip registre mémoire instructions	✓

À la lecture du tableau 3.1, on remarque que peu d'approches analysent le comportement du code avec une méthode d'analyse symbolique (colonne AC, mention Sym). Ce type de méthode à l'avantage de permettre l'analyse de l'effet d'une faute sur un programme

indépendamment de la valeur des entrées. Goubet *et al.* [GHEDK16] sont les seuls à proposer une approche qui analyse le comportement de code bas niveau avec une méthode symbolique en présence de faute transitoires affectant à la fois le code et les données. Néanmoins, l'approche nécessite que le code à analyser soit traduit en un format propre, ne lui permettant pas de prendre en compte l'encodage binaire. Elle ne supporte que des flots de contrôle simples, limitant fortement les codes que l'on peut analyser. Les cases mémoires accédées durant l'exécution du code doivent être déduites du code par l'utilisateur et données en entrées de l'outil. Aussi, cette méthode n'intègre pas de moyen d'analyser le code selon des entrées réalistes et ainsi elle est sujette aux faux positifs. Enfin, l'approche ne propose pas de métrique visant à aider le concepteur de protections ou l'utilisateur en général.

3.4 Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art des différentes approches pour l'analyse des vulnérabilités d'un code. Nous avons d'abord mis en avant les avantages et inconvénients des différentes approches à la modélisation de l'effet des fautes ainsi que des méthodes d'analyses de comportement du code. Puis, nous avons présenté l'état de l'art de l'analyse de robustesse contre des attaques en fautes et nous avons relevé le manque d'approches combinant une méthode d'analyse de comportement du code à bas niveau et symbolique. Toutefois, bien qu'elle soit imparfaite, l'approche proposée par Goubet *et al.* couvre la plus part des critères recherchés ; elle constitue le point de départ de cette thèse.

Dans le prochain chapitre, nous présentons une méthode permettant l'analyse de vulnérabilités de codes bas niveau contre des fautes transitoires altérant à la fois le code et les données. Cette méthode combine exécution symbolique et méthodes formelles pour une analyse de codes aux flots de contrôle complexes et tenant compte de paramètres d'entrée réalistes. Nous proposons aussi de synthétiser les résultats des analyses au travers de trois métriques afin d'aider le concepteur de contre-mesures à se focaliser en priorité sur les portions de codes les plus vulnérables.

Chapitre

4 *Analyse formelle et automatique de robustesse d'un programme binaire contre des attaques en faute*

Sommaire

4.1 Introduction

Nous présentons ici le résultat du travail qui est au cœur de cette thèse : une méthode, et son implémentation dans un outil, visant à l'évaluation automatique et contextuelle de la robustesse de programme binaire en présence de fautes, basée sur une combinaison d'exécution symbolique et de méthodes formelles.

Dans ce chapitre, nous commençons par présenter brièvement la méthode d'analyse de robustesse mise au point. Puis, nous présentons une vue d'ensemble des différentes étapes d'analyses. Nous détaillons ensuite les modèles et comment ils sont employés par la méthode. Nous introduisons ensuite trois métriques synthétisant les résultats de l'analyse au niveau du programme ou mettant en lien les vulnérabilités avec les instructions du code. Ensuite, nous présentons `RobustB`, l'outil dans lequel nous avons implémenté notre méthode et avec lequel nous avons réalisé les expériences présentées dans le chapitre 5 puis nous mettons en relief les éléments spécifiques de notre approche au regard d'autres outils de l'état de l'art.

4.2 Principe de l'approche

L'approche développée dans ce travail de thèse a pour objectif de déterminer si un programme binaire est robuste ou non à des injections de faute. Elle combine des analyses statiques, de l'exécution symbolique et de la vérification formelle pour rechercher la présence de vulnérabilités dans un programme binaire et évaluer la robustesse du code contre des attaques en faute.

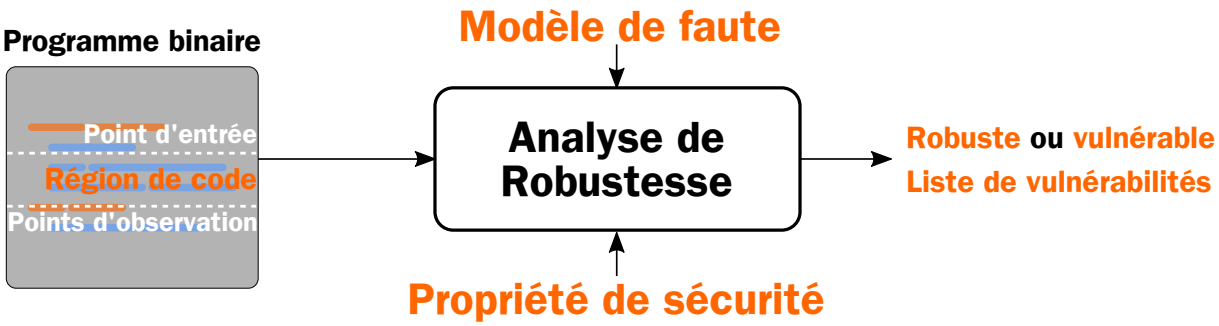

FIGURE 4.1: Interface de l'approche

La figure 4.1 présente l'interface de l'approche : étant donné un programme binaire, une

région de code déterminée par un unique point d'entrée et des points de sortie, l'approche vérifie si la région de code est robuste à certains effets possibles de faute exprimés sous la forme d'un modèle de faute. Déterminer la vulnérabilité de la région de code à une faute nécessite d'avoir un moyen de déterminer si le comportement en présence de faute est acceptable pour la sécurité. C'est le rôle de la propriété de sécurité : celle-ci indique les éléments mémorisant [1] (et le cas échéant leurs relations et/ou leur contenu) permettant de statuer sur le bon comportement de la région de code en un point de celle-ci appelé point d'observation. Il n'y a généralement qu'un point de sortie qui est confondu avec le point d'observation. Toutefois, il se peut que la région de code ait plusieurs points de sortie. Dans ce cas, le point d'observation est confondu avec un des points de sortie de la région de code. Sauf indication contraire, nous considérons dans la suite que la région de code n'a qu'un point de sortie qui est donc confondu avec le point d'observation. La méthode proposée vérifie si les exécutions possibles, c'est-à-dire si les séquences d'instructions allant de l'entrée de la région de code jusqu'au point d'observation peuvent invalider la propriété en présence de faute.

À l'issue de l'analyse de robustesse, la région de code est vulnérable si au moins une injection de faute selon un modèle de faute a invalidé la propriété de sécurité, ou robuste si aucune injection du modèle de faute n'a invalidé la propriété de sécurité et, nous verrons comment dans la section 4.5.4, que la complétude de l'analyse est garantie si l'analyse a exploré tous les cas possibles ou s'il reste des comportements non investigués.

La recherche de vulnérabilité repose sur de la vérification formelle, elle est exprimée sous la forme d'un problème d'equivalence-checking. L'equivalence-checking permet de comparer si deux modèles (ici, un modèle correspondant à une exécution de la région de code sans faute et un autre correspondant à une exécution de la région de code en présence de faute) présente le même comportement. Cette technique permet d'exprimer des propriétés de sécurité en faisant référence au modèle non fauté et évite ainsi d'avoir à caractériser et formaliser les comportements corrects possibles de la région de code.

Les méthodes formelles sont connues pour être sujettes à l'explosion combinatoire, ce qui limite la taille et la complexité des problèmes qui peuvent être résolus. Lors de la mise au point de notre approche, nous avons cherché à repousser l'explosion combinatoire en diminuant la taille et la complexité des problèmes analysés. Nous avons plus précisément cherché à réduire la taille des modèles représentant les exécutions possibles de la région de code. Nous avons ainsi choisi de modéliser formellement qu'une partie du comportement de la région de code à la fois : nous vérifions la vulnérabilité de la région de code face à une faute pour chaque chemin d'exécution de la région de code indépendamment. Cela permet

1. Élément mémorisant : registre ou case mémoire accessible explicitement par le logiciel.

de découper le problème initial en plusieurs problèmes plus petits. Une autre difficulté, introduite dans le chapitre précédent est celui des potentiels faux positifs des approches symboliques. Pour notre approche, reposant sur une méthode symbolique, cette difficulté se pose d'autant plus car nous analysons un morceau de code se situant à l'intérieur d'un programme : les valeurs possibles de tous les éléments mémorisants qui sont accédés lors de l'exécution de la région de code dépendent du chemin d'exécution qui permet d'arriver à l'entrée de la région de code. De façon à circonscrire les valeurs possibles des éléments mémorisants, nous avons choisi d'utiliser, en amont de la recherche de vulnérabilité, une analyse du programme basée sur de l'exécution symbolique pour déterminer les ensembles de valeurs possibles en entrée de la région de code lors d'exécutions réelles du programme.

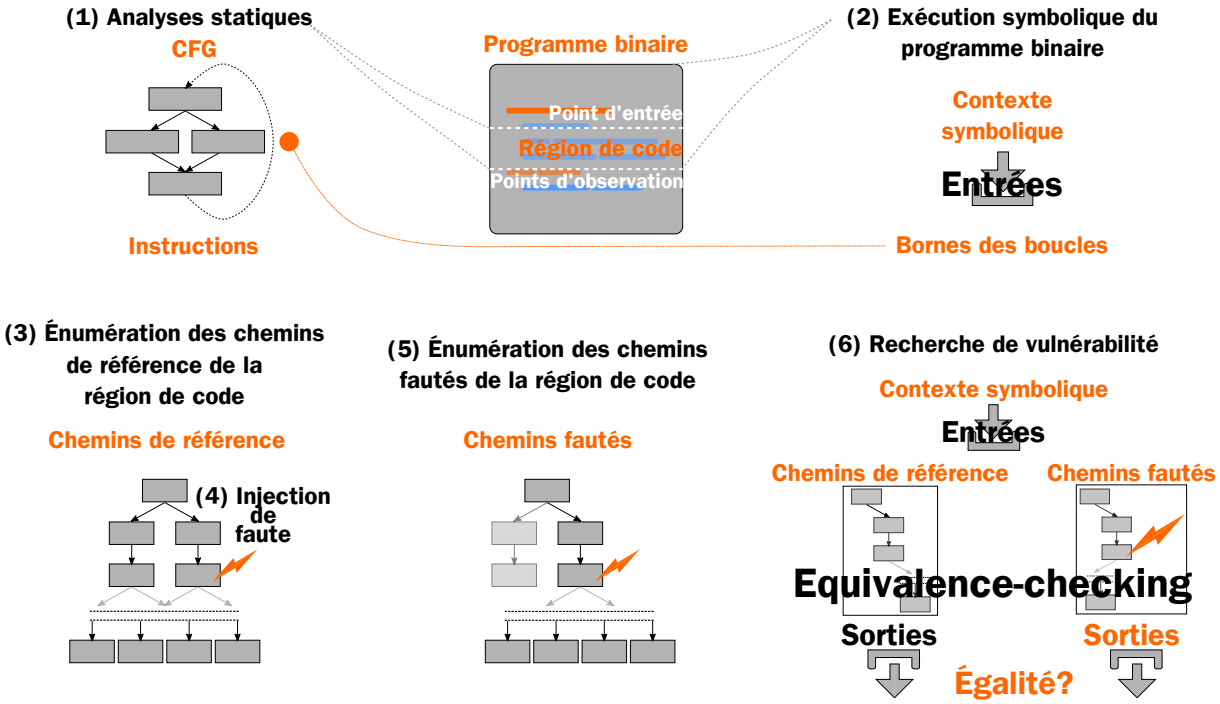

FIGURE 4.2: Éléments de l'approche

La figure 4.2 illustre les différentes étapes de l'analyse de robustesse proposée ainsi que les éléments y entrant en jeu. L'approche comporte six étapes. Les deux premières sont : (1) l'analyse statique du programme et de la région de code (2) l'exécution symbolique du programme binaire partant du début du programme jusqu'au point de sortie de la région de code. Le but de ces analyses est d'obtenir quatre éléments : le graphe de flot de contrôle (CFG) de la région de code, les instructions de la région de code (assembleur et encodage), un contexte symbolique de la région de code ainsi que les bornes de boucles de la région de code. Un contexte symbolique décrit symboliquement l'ensemble des configurations possibles des éléments mémorisant en entrée de la région de code. Ce contexte, obtenu par

l'exécution symbolique du programme binaire, est utilisé pour restreindre les vulnérabilités trouvées à celles qui sont possibles considérant les configurations d'entrées autorisées par le contexte symbolique. Ces dernières correspondent à des valeurs possibles lors d'une exécution de code réelle et permettent donc de limiter les faux positifs.

Les quatre éléments (CFG, instructions, bornes des boucles, contexte symbolique) obtenus par les deux premières étapes sont utilisés dans les étapes suivantes de l'approche pour déterminer les chemins d'exécution avec et sans faute dans le but de modéliser la recherche de vulnérabilité. Dans la suite, on appelle chemin une séquence d'instructions qui peut être exécutée et qui part du point d'entrée de la région de code et termine au point de sortie dans la région de code. L'énumération des chemins possibles sans faute, appelés chemins de référence, détermine l'ensemble des chemins de référence à partir du CFG, des bornes des boucles et du contexte symbolique (3). Les effets des fautes sont modélisés sur les chemins de référence (4) et les chemins fautés sont énumérés à partir des instructions, du CFG, des bornes de boucles et du contexte symbolique en tenant compte de ces effets (5). Un chemin fauté est toujours associé à un chemin de référence, cela est nécessaire pour modéliser la recherche de vulnérabilité (6). En effet, la recherche de vulnérabilité est réalisée par equivalence-checking et repose sur une propriété de sécurité mettant en relation des éléments mémorisant du chemin fauté avec leurs équivalents dans le chemin de référence associé.

La figure 4.3 décrit plus précisément la recherche de vulnérabilité. Elle est formulée comme un problème SMT qui fait entrer en jeu quatre éléments : un chemin fauté, le chemin de référence associé au chemin fauté, un contexte symbolique et la propriété de sécurité. La résolution du problème SMT cherche s'il existe une configuration des entrées autorisée par le contexte symbolique qui invalide la propriété de sécurité et qui permet la faisabilité du chemin de référence et du chemin fauté. Comme déjà dit, une propriété de sécurité d'équivalence établit une relation d'égalité entre le contenu de certains éléments mémorisants du chemin fauté et du chemin de référence au point d'observation. Une vulnérabilité est détectée si le contenu d'au moins un élément mémorisant, référencé dans la propriété de sécurité, est différent dans le chemin de référence et dans le chemin fauté. L'approche est construite pour permettre d'exprimer la propriété comme une *propriété d'équivalence*. Toutefois, il est possible d'exprimer des propriétés qui ne font pas référence aux éléments mémorisant du chemin de référence. Ces propriétés sont appelés *propriétés spécifiques*. Elles ne font référence qu'aux éléments mémorisant du chemin fauté. Dans ce cas, plutôt que de comparer les contenus d'éléments mémorisant du chemin fauté avec ceux du chemin de référence, ces propriétés explicitent les contenus ou relations attendus pour certains éléments mémorisants du chemin fauté au point d'observation. Une vulnérabilité est détectée lorsque les relations entre ces éléments mémorisants du chemin fauté sont

invalidées ou si leurs contenus diffèrent de ceux attendus.

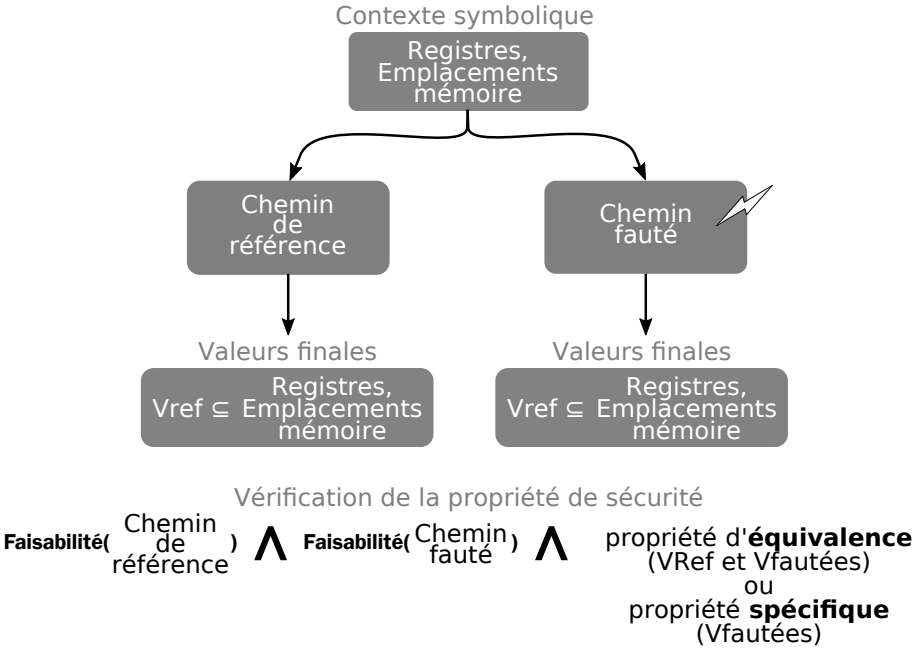

FIGURE 4.3: Analyse de robustesse

La figure 4.4 présente l'approche dans son ensemble. Celle-ci a été implémentée dans un outil, `RobustB`. On y retrouve les quatre éléments qui doivent être fournis à l'outil. Le premier est un fichier binaire, au format ARM Thumb-2 [2]. Le jeu d'instructions ARM Thumb-2 a été pensé pour réduire l'empreinte mémoire faisant de lui un bon candidat dans le domaine des systèmes embarqués, particulièrement exposé aux attaques en faute. Le second est la région de code à analyser dans le fichier binaire. Le troisième élément est le modèle de faute vis-à-vis duquel la région de code doit être analysée. Ce modèle de faute est appliqué le long de chaque chemin de référence. Enfin, le quatrième élément est la propriété de sécurité. Dans un premier temps, `RobustB` extrait des informations du fichier binaire par une analyse statique et une exécution symbolique du programme binaire avec l'aide d'un outil externe (`angr`). Vient ensuite le cœur de l'approche qui opère sur une représentation de la région de code pour énumérer les chemins de référence, y injecter les fautes et énumérer les chemins fautés avec l'aide d'un solveur SMT externe (`Z3`). L'analyse de robustesse est ensuite réalisées, elle consiste en la construction d'autant de problèmes SMT que nécessaire à la recherche de vulnérabilités pour chaque occurrence du modèle de faute sur chaque chemin de référence avec l'aide du solveur SMT externe. L'analyse de

2. ARM Thumb-2 : Thumb-2 est une extension du jeu d'instructions de 16 bits Thumb de Arm. Il enrichit le jeu d'instructions Thumb avec des instructions 32 bits faisant de lui un jeu d'instructions à taille variable.

robustesse aboutit à un ensemble de vulnérabilités. Cet ensemble peut-être vide mais aussi très grand en fonction de la sensibilité de la région de code au modèle de faute considéré. Lorsque l'analyse révèle des vulnérabilités, celles-ci sont synthétisées en trois métriques : deux d'entres elles indiquent un score global sur la sensibilité de la région de code et la dernière exprime la sensibilité de chaque instruction de la région de code.

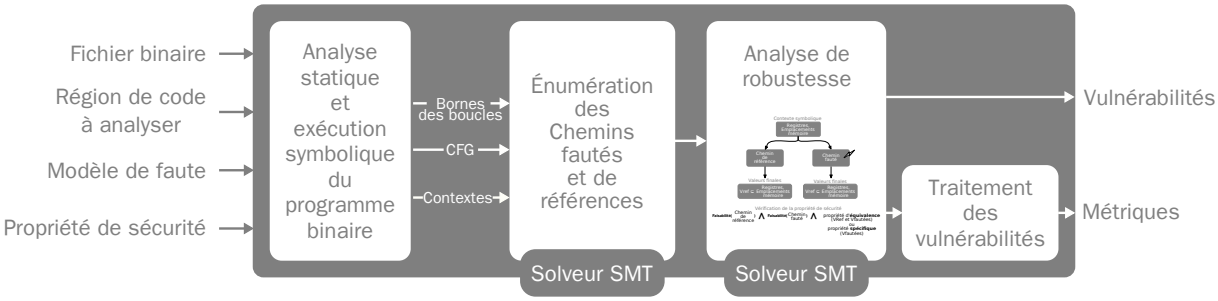

FIGURE 4.4: Vue d'ensemble de l'approche

La partie suivante détaille les différentes analyses de l'approche réalisées sur le programme binaire permettant d'obtenir les informations nécessaires à la représentation de la région de code ainsi que les contextes symboliques et les bornes de boucles.

4.3 Analyse du programme binaire

L'analyse du programme binaire a pour but d'obtenir les informations nécessaires à la représentation de la région de code d'une part et d'obtenir les contextes symboliques et les bornes de boucles associées d'autre part.

Une représentation de la région de code peut-être construite grâce à son CFG et ses instructions assembleur. Le CFG de la région de code est obtenu par une analyse statique du programme binaire tandis que les instructions assembleur sont obtenues à l'aide d'un chargeur (*loader*) de binaire et un désassembleur pour le second.

Les parties suivantes présentent comment sont obtenus les bornes de boucles et les contextes symboliques.

4.3.1 État symbolique du programme binaire en entrée de la région de code

L'état symbolique du programme binaire en entrée de la région de code est une étape intermédiaire à la construction du contexte symbolique. Il est obtenu par une première exécution symbolique, nommée extraction de l'état symbolique du programme en entrée de la région. Elle a pour but de déterminer les différentes configurations possibles des éléments mémorisants, chacune correspondant à un chemin d'exécution menant à la région de code dans le binaire. Ce sont les configurations des éléments mémorisants extraits de cette analyse qui sont appelées *états symbolique du programme*. Un état symbolique du programme est un ensemble de variables dont les valeurs sont symboliques et où chacune représente un élément mémorisant. Il y a deux avantages à une telle démarche ; d'abord, cela permet de réduire l'espace des configurations des entrées à explorer lors de la résolution des problèmes SMT et ainsi diminuer la durée de résolution des problèmes. Cela est particulièrement vrai lorsque le problème n'a pas de solution, *i.e.* lorsque la faute ne met pas en évidence une vulnérabilité. Dans ce cas, le solveur énumère toutes les valeurs possibles et la présence d'un état symbolique du programme, réduisant le nombre de valeurs possibles, peut permettre au solveur de répondre là où il ne l'aurait autrement pas pu. Le second apport de cette démarche est l'élimination de faux positifs. En effet, initialiser les registres et les cases mémoire avec un état symbolique du programme limite les valeurs possibles des cases mémoire et registres utilisés à des valeurs possible lors d'une exécution de code réelle et permet donc d'éliminer d'éventuelles configurations impossibles. Ces états symboliques sont extraits par une exécution symbolique du code. L'exécution symbolique exécute le programme en symbolisant ses entrées (normalement fournies par l'utilisateur), elle a pour but d'atteindre tous les points du programme menant au point d'entrée de la région de code à analyser et d'obtenir, en ces endroits, les expressions symboliques associées aux registres et aux cases mémoire accédées depuis le début du programme. À l'issue de cette analyse, toutes les expressions symboliques de tous les registres et emplacements mémoire aux différents points du programme menant au point d'entrée de la région de code sont extraites.

4.3.2 Contextes symboliques et bornes de boucle

La construction du contexte symbolique nécessite une autre exécution symbolique. Elle concerne cette fois la région de code elle même. Celle ci, nommée extraction du contexte symbolique de la région de code et des bornes de ses boucles, a pour but de déterminer la borne supérieures des itérations des boucles pour chaque état symbolique du programme

obtenu lors de l'extraction de l'état symbolique du programme en entrée de la région de code. En effet, cette analyse peut extraire des intervalles de valeurs pour les bornes des boucles. Dans ce cas, toutes les itérations jusqu'à l'atteinte des bornes supérieures des boucles sont considérées lors du déroulage du CFG. Il se peut que cette analyse extrait des bornes de boucles symboliques, dans ce cas l'approche ne pourra pas analyser l'intégralité des itérations et ne pourra donc pas conclure sur la robustesse de la région de code. Cette analyse permet aussi de réduire la complexité des problèmes SMT en limitant les cases mémoire représentées dans les états symboliques du programme à celles qui sont accédées par la région de code. Dans cette analyse, la région de code est exécutée symboliquement pour chaque état symbolique du programme. À l'issue de l'exécution, l'état symbolique du programme est privé des cases mémoire qui n'ont pas été accédées durant l'exécution de la région de code pour former un *contexte symbolique*. Plus formellement, les éléments mémorisants présents dans un contexte symbolique Co sont définis par la formule suivante :

$$Co = r \cup (EMR^m \cap EMP^m)$$

où r désigne l'ensemble des registres accessibles du logiciel, EMR^m (pour Éléments Mémorisant accédées par la Région de code) est l'ensemble des emplacements mémoire (^{m}) accédés lors de l'exécution de la région de code (obtenu lors de l'extraction de l'état symbolique du programme en entrée de la région) et EMP^m est l'ensemble des emplacements mémoire accédés lors de l'exécution symbolique du programme juste avant le point d'entrée de la région de code (obtenu lors de l'extraction du contexte symbolique de la région de code et des bornes de ses boucles). Chaque élément d'un Co est associé à une expression symbolique représentant les valeurs possibles de cet élément, obtenues lors de la première analyse dynamique du binaire, correspondant à celles possibles juste avant le point d'entrée de la région de code.

4.4 Énumération des chemins de référence et des chemins fautés de la région de code

Cette partie présente l'énumération des chemins de référence et des chemins fautés. Les chemins de référence sont obtenus par le déroulage du CFG jusqu'au point d'observation en considérant les bornes des boucles. Les fautes correspondant au modèle de faute spécifié sont ensuite injectées le long des chemins de références, et les chemins fautés résultant en sont déduits en prenant en compte leurs effets potentiels sur le flot de contrôle.

4.4.1 Énumération des chemins de référence

Les chemins de références correspondent aux exécutions possibles de la région de code étant donné un de ses contextes symboliques. Ils sont déterminés par déroulage du CFG couplé avec une analyse de faisabilité (cf. section 4.5).

Le déroulage des boucles du CFG est réalisé étant donné les bornes supérieures des boucles de la région de code associées à un contexte symbolique. Les chemins issus du déroulage du CFG commencent par l'instruction correspondant au point d'entrée de la région de code et se terminent par l'instruction correspondant au point d'observation de la propriété. Le déroulage du CFG est réalisé statiquement : seuls les blocs, les arcs et les bornes supérieures des arcs arrières (*i.e.* le nombre de fois que les arcs arrières peuvent être pris) sont considérés. L'ensemble des chemins issus du déroulage du CFG peut être représenté comme un arbre dont la racine est le bloc correspondant au point d'entrée de la région de code et dont les feuilles sont des blocs associés au point d'observation. Dans la suite, nous appelons cette représentation *PEFG* pour *Potential Execution Flow Graph*. N'évaluant pas les instructions lors de la construction du PEFG, il n'y a pas de garantie sur la faisabilité des chemins contenus dans le PEFG compte tenus des contextes symboliques. Une analyse de faisabilité est nécessaire pour déterminer quels sont ceux réellement possibles pour les configurations des entrées possibles dictées par les contextes symboliques.

La faisabilité d'un chemin est déterminée par vérification formelle de la conjonction de formules logiques représentant les instructions le long du chemin initialisé par le contexte symbolique. Cette analyse est présentée dans le détail dans la partie 4.5.2.

4.4.2 Modèles de faute

Une faute est injectée en un point le long d'un chemin de référence. Chaque faute est définie par un modèle de faute et celui-ci définit les points du programme auxquels le modèle de faute peut être appliqué ainsi que les éléments qu'il affecte. Par exemple, le modèle de faute de corruption de registre ne peut être appliqué qu'à un registre entre deux instructions tandis que le modèle de faute de saut d'instruction ne peut être appliqué qu'à une instruction. Les modèles de faute supportés sont listés ci-dessous et sont suivis d'une description des modalités de leurs injections.

La notion de point d'injection doit être définie pour chaque modèle de faute pour pouvoir injecter la faute. Un point d'injection décrit une adresse et une occurrence de

l'adresse dans un chemin et l'élément (instruction ou registre) affecté par la faute. Par exemple, pour le modèle de faute de saut d'instruction, un point d'injection est une instruction le long d'un chemin. Pour le modèle de faute de corruption de registre, un point d'injection correspond à n'importe quel registre ou drapeau entre deux instructions le long d'un chemin.

saut d'instruction : ce modèle de faute s'applique à toutes les instructions le long d'un chemin de référence. Il a pour effet de supprimer la réalisation des opérations de l'instruction visée. Un tel effet a pu être observé de nombreuses fois [MDH+13, BGV11, KMW17, BJC15].

corruption de registre : ce modèle de faute s'applique à tous les registres (à l'exception du compteur ordinal, du pointeur de pile et du registre contenant l'adresse de retour) entre deux instructions d'un chemin de référence. Le choix de la valeur du registre visé est laissé libre au solveur SMT. Lors de la résolution du problème SMT, le solveur SMT peut explorer l'ensemble des valeurs du registre visé pour résoudre le problème. Ce modèle de faute englobe des effets de fautes obtenus avec différents moyens d'injection sur différents éléments de la microarchitecture. Par exemple, mise à 0 ou mise à 1 d'un bit d'un registre [BECN+06] ou encore faute lors d'un transfert d'une donnée sur le bus [MDH+13].

remplacement d'instruction : ce modèle de faute s'applique à toutes les instructions et à tous les registres après l'instruction visée. Il a pour effet de supprimer la réalisation des opérations de l'instruction visée et de corrompre un registre (autre que le compteur ordinal, le pointeur de pile et le registre contenant l'adresse de retour). Ce modèle de faute émule le remplacement d'une instruction par une autre instruction ayant potentiellement d'autres opérandes, un autre registre destination et réalisant une autre opération. Par exemple, l'effet de ce modèle de faute peut être obtenu par injection électromagnétique visant le bus d'instruction pendant un transfert [MHER14] ou par clock glitches [BGV11].

instruction bit-set : ce modèle de faute s'applique à tous les bits à 0 de toutes les instructions. Il a pour effet de mettre à 1 un bit dans le codage d'une instruction. Cet effet a été observé en fautant des éléments de la mémoire Flash lors de la lecture d'une instruction [CMD+18].

instruction bit-reset : ce modèle de faute s'applique à tous les bits à 1 de toutes les instructions. Il a pour effet de mettre à 0 un bit dans le codage d'une instruction. Cet effet a été observé en fautant le bus d'instruction lors d'un transfert [RSDT13, KBB+18].

Il est a priori possible de représenter n'importe quel modèle de faute tant que ses effets

sont observables au niveau du CFG (effet sur le flot de contrôle), des instructions ou des éléments mémorisant accessibles du logiciel. Toutefois, les points d'injections et les effets de certains modèles de faute, notamment architecturaux, sont difficilement déductibles sans informations concernant l'architecture. D'autres modèles de faute, comme la corruption directe du compteur ordinal ne peuvent pas, en pratique, être considérés exhaustivement. La corruption du compteur ordinal revient à permettre à l'exécution de continuer à partir d'une adresse aléatoire. Outre les problèmes que cela pose dans un jeu d'instructions à taille variable, réaliser ce type de faute de façon exhaustive [3] fait croître le nombre de chemins fautés et présente peu d'intérêt en force brute.

Aussi, dans les modèles de faute instruction bit-set et instruction bit-reset, la faute n'est pas réalisée lorsque le nouveau codage de l'instruction est interprété comme un saut (alors qu'il ne l'était pas originellement) ou que le modèle de faute modifie l'adresse d'un saut. La modélisation des modèles de faute instruction bit-set et instruction bit-reset pourraient prendre en compte un saut vers une instruction mais, par manque de temps, cela n'a pas été réalisé ; ce type de faute n'est donc pas pris en compte dans les expériences présentées dans le chapitre 5.

4.4.3 Énumération des chemins fautés

Les chemins fautés sont déterminés à partir des chemins de références et d'un modèle de faute. Une faute, injectée le long d'un chemin de référence, peut faire dévier le chemin suivi après la faute par rapport au chemin de référence. Par exemple, sauter une instruction de branchement force l'exécution à emprunter le bloc suivant en séquence ; corrompre un registre peut changer le résultat d'un branchement conditionnel. En conséquence, les chemins fautés possibles doivent être recalculés, en utilisant le CFG, à partir du bloc où la faute a été injectée. On nomme ce bloc *BBfauté*. Si la faute a pour effet de changer le flot de contrôle (*i.e.* sauter d'une instruction de branchement inconditionnel), alors le chemin partant du bloc racine jusqu'au BBfauté est relié au CFG puis le CFG est déroulé. Si la faute n'a pas d'effet direct sur le flot de contrôle, celui-ci peut tout de même être indirect, comme lors d'une modification de la valeur d'une variable utilisée dans un test conditionnel. Pour capturer ces effets, le préfixe du chemin fauté partant du point d'entrée de la région de code jusqu'au BBfauté, inclus, est relié au CFG, puis l'ensemble est déroulé pour former un PEFG fauté. La faisabilité des chemins du PEFG est ensuite testée. Les chemins faisables du PEFG fauté forment l'ensemble des chemins fautés associé à une même faute sur un chemin de référence.

3. nombre de fautes possibles : $\sim$ (*nombre moyen d'instructions par chemin* $\times$ *nombre de chemins* $\times$ *nombre d'instructions de la région de code*)

Une faute peut aussi modifier le nombre d'itérations d'une boucle. Pour prendre en compte ce cas, les bornes des boucles sont relâchées lors du déroulage du CFG après l'injection de la faute ; par défaut, une itération supplémentaire au-delà des bornes de boucles initiales est réalisée. Le nombre de tours de boucle supplémentaire n'étant a priori pas connu dans le cas général, et potentiellement très grand, il n'est pas envisageable de dérouler la boucle jusqu'à ce qu'une analyse de faisabilité révèle la nouvelle borne de boucle induite par la faute. Il est toutefois intéressant de détecter qu'une faute induit plus d'itérations que celles initiales. Pour cela, nous définissons un nouvel ensemble de chemins, les chemins frontières, constitué de préfixe de chemin résultant d'une itération additionnelle au-delà des bornes de boucles considérées. Un chemin frontière commence à la première instruction de la région de code et se termine à la première instruction de l'itération suivante d'une boucle. Les chemins frontières sont utilisés, à travers leur faisabilité, pour déterminer la complétude de l'énumération des chemins fautés (cf. section 4.5.4).

4.5 Modélisation du problème de robustesse et des éléments qui le composent

Cette partie présente la représentation formelle d'un chemin qui est utilisée dans les différentes analyses et, en particulier, comment la faisabilité d'un chemin peut être décidée à partir de sa représentation formelle et d'un contexte symbolique extrait de l'analyse du programme binaire. Puis, la représentation formelle des fautes est expliquée ainsi que l'analyse déterminant la complétude de l'énumération des chemins fautés et l'analyse de robustesse. Enfin, nous interprétons les résultats des deux analyses vis-à-vis d'une faute sur un chemin de référence.

Un chemin φ peut être décrit à partir de la séquence d'instructions qui le compose. Chaque instruction est formellement représentée et la représentation formelle d'un chemin est la conjonction des représentations formelles des effets des instructions qui composent le chemin, sur l'état global du programme. Dans la suite, nous appelons $TF(\varphi)$ cette représentation formelle d'un chemin φ.

4.5.1 Représentation formelle d'un chemin

La représentation formelle d'un chemin φ, nommée $\mathrm{TF}(\varphi)$, composé de n instructions est une formule logique comprenant les variables correspondants aux éléments mémorisants après l'exécution d'une instruction d'un chemin et avant l'exécution de la première

instruction. Plus formellement, on définit :

— V l'ensemble des variables de la formule TF. Chaque élément de V contient une expression symbolique représentant les états possibles de l'élément mémorisant correspondant après l'exécution d'une instruction ou avant l'exécution de la première instruction dans la formule TF. V peut être exprimé comme l'union des ensembles V_i, où chaque $V_i (i \in [1..n])$ contient les variables représentant les éléments mémorisants après l'exécution de l'instruction i, et V_0, représente les expressions symboliques des éléments mémorisants avant l'exécution de la première instruction :

$$V = \cup_{i \in [0..n]} V_i$$

n représente ici le nombre d'instructions du chemin. On définit par ailleurs :

— $D(v)$ le domaine de définition de la variable $v \in V$.

— $S_i \subseteq \prod_{v \in V_i} D(v)$ l'ensemble des configurations possibles des variables de V_i.

— $TR_i \subseteq S_{i-1} \times S_i$ les configurations possibles des variables dans $V_{i-1} \cup V_i$ étant données les contraintes dérivées de la sémantique de la i^{ime} instruction.

— $\chi_E : E \to \{0, 1\}$ la fonction caractéristique d'un ensemble E. [4]

De manière informelle, le modèle d'une instruction i est une relation contraignant les variables dans l'ensemble V_i, en accord avec la sémantique de l'instruction i, considérant le contenu des variables dans l'ensemble V_{i-1}. La représentation formelle TF(φ) dérivée d'un chemin φ est définie par la formule suivante :

$$TF(\varphi) = \bigwedge_{i=1...n} \chi_{TR_i}$$

De manière informelle, V_0 fixe les valeurs initiales possibles des éléments mémorisants, TR_1 utilise ses valeurs et contraint les variables dans l'ensemble V_1 étant donnée l'opération réalisée par la première instruction. TR_2 utilise V_1 pour contraindre les variables dans l'ensemble V_2 et ainsi de suite jusqu'à la $n^{ième}$ instruction. L'ensemble V_n contient alors les valeurs possibles des éléments mémorisants étant données les valeurs initiales possibles contenues dans V_0 et la sémantique des instructions du chemin.

4. Fonction caractéristique : ensemble de (k+1)-tuples, tel que $\chi_E(v_0, \ldots, v_k) = 1$ si $(v_0, \ldots, v_k) \in E$ et 0 sinon.

4.5.2 Faisabilité

La faisabilité d'un chemin représenté par la formule TF détermine si le chemin correspondant est faisable pour un contexte symbolique donné. Un chemin φ est faisable si la formule suivante est satisfiable :

$$Feas(\varphi, ctx) = \chi_{S_0} \wedge \bigwedge_{i=1...n} \chi_{TR_i}$$

Dans cette formule, χ_{S_0} est la fonction caractéristique de S_0 qui exprime les configurations possibles des variables initiales V_0 reflétant le contenu du contexte symbolique ctx, obtenu par l'analyse dynamique du binaire.

Un chemin est faisable étant donné un contexte symbolique s'il existe une configuration des variables d'entrée pour laquelle les conditions des directions des branchements conditionnels le long de ce chemin peuvent être satisfaites. Autrement dit, un chemin n'est faisable que si chaque branchement conditionnel le long de ce chemin peut être pris ou non pris en fonction de la séquence de blocs qui compose le chemin. Pour émuler le comportement des branchements conditionnels, ils sont modélisés de façon à contraindre les variables déjà contraintes par l'instruction précédente. De cette manière, si les valeurs possibles dans l'ensemble V_k d'une instruction k sont en conflit avec les contraintes ajoutées par l'instruction i_{k+1} correspondant au branchement conditionnel suivant l'instruction, alors la formule TF n'est pas satisfiable et le chemin est considéré comme infaisable.

Nous illustrons la modélisation d'un chemin et de sa faisabilité par un exemple dont le code est présenté dans le listing 4.1.

Ce court programme assembleur contient deux chemins : si les registres `r0` et `r1` contiennent la même valeur alors le branchement conditionnel est pris et le registre `r0` est incrémenté ; sinon, le branchement conditionnel n'est pas pris et `r0` est incrémenté de deux.

```
        cmp     r0, r1          % a
        b.eq    label           % b
        add     r0, r0, #1      % c
label:  add     r0, r0, #1      % d
```

Listing 4.1: Code assembleur représentant deux chemins (illustration des modélisations d'un chemin et de sa faisabilité)

On note $\varphi 0$ et $\varphi 1$ les deux chemins possibles, $\varphi 0$ correspond à la séquence d'instructions $\{a,b,c,d\}$ et $\varphi 1$ à la séquence $\{a,b,d\}$.

Une illustration de la formule TF associée à $\varphi 0$ et $\varphi 1$ est donnée à la figure 4.5.

L'ensemble $V_i, i \geq 1$ contient les variables contraintes par l'instruction i. V_0 contient l'ensemble initial des variables non contraintes. Dans une analyse de faisabilité, V_0 est contraint par un contexte symbolique issu de l'analyse du programme binaire.

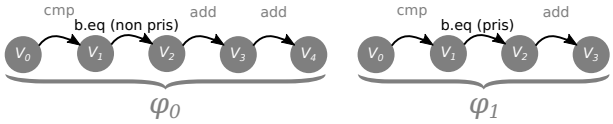

FIGURE 4.5: Exemple pour l'illustration des formules TF(φ0) et TF(φ1) avec φ0 et φ1 les deux chemins possibles dans le listing 4.1

Pour chaque chemin φ, la formule TF(φ) est composée des contraintes induites pour chaque instruction i contenue dans la séquence d'instructions du chemin. Dans la suite, on note $nom\_de\_variable_i$ une variable contenue dans l'ensemble V_i. Les contraintes sur les variables dans l'ensemble V_i qui ne sont pas issues d'une interprétation sémantique d'une instruction ne sont pas représentées : lorsqu'une instruction i n'impacte pas un élément mémorisant, les variables des ensembles V_i et V_{i-1} représentant ces éléments sont simplement mis en relation de façon à transmettre la valeur des éléments mémorisants correspondant depuis l'ensemble V_{i-1} vers l'ensemble V_i : $var_i = var_{i-1}$.

	$TF(\varphi 0)$:	$TF(\varphi 1)$:
(cmp)	$F.Z_1 = r0_0\ eq\ r1_0$	$F.Z_1 = r0_0\ eq\ r1_0$
$(b.eq)$	$\wedge\ F.Z_2 = F.Z_1 \wedge F.Z_2 = \mathbf{1}$	$\wedge\ F.Z_2 = F.Z_1 \wedge F.Z_2 = \mathbf{0}$
(add)	$\wedge\ r0_3 = r0_2 + 1$	$\wedge\ r0_3 = r0_2 + 1$
(add)	$\wedge\ r0_4 = r0_3 + 1$	

Dans les deux formules, le modèle de l'instruction `cmp` contraint le drapeau Z ($F.Z$, mis à '1' lorsque le test *eq* réussit) dans l'ensemble V_1 à valoir '1' si les registres `r0` et `r1` dans V_0 sont égaux et sinon à la valeur '0'. En plus de la propagation de la valeur du drapeau Z, le modèle de l'instruction `b.eq` ajoute une contrainte au drapeau Z dans l'ensemble V_2 signifiant son égalité avec la valeur nécessaire pour que ce chemin soit emprunté. Lors de l'évaluation de la faisabilité du chemin φ0, *i.e.* quand les variables dans l'ensemble V_0 sont contraintes par le contexte symbolique, si les configurations possibles ne permettent pas aux registres `r0` et `r1` d'être différents, alors la formule correspondant à la modélisation de l'instruction `b.eq` sera forcément non satisfiable et le chemin φ0 sera indiqué comme infaisable.

4.5.3 Modélisation des fautes

Les effets des injections de faute doivent être modélisés dans la formule TF d'un chemin fauté lors de l'analyse de faisabilité des chemins fautés. L'injection d'une faute à l'instruction j d'un chemin de référence de n instructions revient à remplacer ou transformer la $j^{ème}$ transition par une modélisation de l'effet de la faute :

$$TF(\varphi_r, ctx) = \bigwedge_{i=1...n} \chi_{TR_i}$$

$$TF(\varphi_f, ctx) = \bigwedge_{i=1...j-1} \chi_{TR_i} \wedge \chi_{TR_{j_f}} \wedge \bigwedge_{i=j+1...m} \chi_{TR_i}$$

où $TF(\varphi_r, ctx)$ est la représentation formelle du chemin de référence et $TF(\varphi_f, ctx)$ est la représentation formelle du chemin fauté. Dans la formule $TF(\varphi_f, ctx)$, la modélisation de l'instruction fautée (j) est modifiée pour prendre en compte l'effet de la faute sur l'instruction. De plus, les instructions suivantes ($[j+1, m]$) peuvent être différentes des instructions $[j+1, n]$ dans la formule du chemin de référence si la faute j impacte le flot de contrôle.

Les modèles de faute instruction bit-set et instruction bit-reset peuvent avoir un effet de bord sur les instructions exécutées après la faute. En particulier lorsque, comme c'est le cas ici, le jeu d'instructions est à taille variable. La modification d'un bit peut alors changer l'interprétation que fera le processeur sur la largeur de l'instruction à exécuter.

La modélisation des différents modèles de fautes dans la formule TF est la suivante :

saut d'instruction : ce type de faute est modélisé par la suppression des contraintes de l'instruction visée. La transition de l'instruction visée est supprimée et remplacée par une autre n'ayant aucun effet, *i.e.* les contraintes des variables issues des instructions précédentes sont transférées aux variables de l'instruction visée par la faute.

corruption de registre : le registre visé devient libre après l'exécution de l'instruction visée : le choix de la valeur de ce registre est laissé au solveur SMT. Ce registre libre est utilisé par les transitions suivantes jusqu'à ce qu'une autre instruction réécrive dans le registre. Les contraintes dérivées de la sémantique de l'instruction visée s'appliquent normalement, exceptées celles exprimant une relation avec le registre visé qui sont supprimées.

remplacement d'instruction : ce modèle de faute est une combinaison du modèle de faute de saut d'instruction et de corruption de registre. Les contraintes des variables de l'instruction précédente sont transférées aux variables de l'instruction fautée

exceptées celle correspondant au registre visé. Ce modèle de faute correspond au remplacement d'une instruction par une autre qui a un registre destination qui n'est ni le compteur ordinal ni le pointeur de pile ni le registre contenant l'adresse de retour d'une fonction.

instruction bit-set : mise à '1' d'un bit de l'instruction visée. L'encodage de l'instruction doit être réinterprété. La transition de l'instruction visée est remplacée par celle correspondant à l'instruction nouvellement interprétée.

instruction bit-reset : mise à '0' d'un bit de l'instruction visée. L'encodage de l'instruction doit être réinterprété. La transition de l'instruction visée est remplacée par celle correspondant à l'instruction nouvellement interprétée.

4.5.4 Analyse de complétude de l'énumération des chemins fautés

Cette partie présente, dans un premier temps, l'algorithme d'énumération des chemins fautés et frontières. Elle présente ensuite l'analyse réalisant la vérification de la complétude de l'énumération des chemins fautés.

L'algorithme 1 montre l'énumération des chemins fautés et des chemins frontières considérant un ensemble de modèles de faute M et un ensemble de chemins de références ϕ_r.

Algorithm 1 Énumération des chemins fautés et des chemins frontières

```
1: procedure ÉNUMÉRATION_DES_CHEMINS(M, φ_r)
                                          ▷ Pour tous les modèles de fautes
2:     for m ∈ M do
                                          ▷ Pour tous les chemins de référence
3:         for φ ∈ φ_r do
                                          ▷ Pour tous les points d'injection
4:             for k ∈ IP_φ^m do
5:                 φ_t_m^k, φ_f_m^k = injection_faute(k, m, φ)
```

IP_{φ}^{m} est l'ensemble des points d'injection du modèle de faute m sur le chemin φ. La fonction $injection\_faute(k, m, \varphi)$ injecte l'occurrence k du modèle de faute m sur le chemin φ et renvoie deux ensembles de chemins :

- $\varphi_{f}{}_{m}^{k}$: l'ensemble de chemins fautés induits par l'injection de l'occurrence k du modèle de faute m sur le chemin de référence φ.
- $\varphi_{t}{}_{m}^{k}$: l'ensemble de chemins frontières induits par l'injection de l'occurrence k du modèle de faute m sur le chemin de référence φ.

Un chemin frontière est un préfixe de chemin commençant à la première instruction de la région de code et qui se termine à la première instruction de la première itération non considérée. La faisabilité d'un chemin frontière indique qu'au moins une itération supplémentaire est réalisable et qu'il existe au moins un chemin contenant au moins une itération supplémentaire. Étant donné que la faisabilité d'un chemin frontière englobe tous les chemins contenant au moins une itération supplémentaire, sa non faisabilité garantit que l'exploration des chemins fautés est complète : il n'y a pas de chemins contenant plus d'itérations que celles considérées. La complétude de l'énumération des chemins fautés n'est utile que si la région de code analysée contient une boucle. En effet, l'analyse détermine si le déroulage des boucles est suffisant, *i.e.* s'il existe des chemins correspondant à plus d'itérations. Pour un code sans boucle, l'énumération des chemins fautés est trivialement complète, *i.e.* il n'y a pas de chemins frontières. Pour une faute, la complétude de l'énumération des chemins fautés est déterminée par les analyses de faisabilité des chemins frontières issus d'une injection de faute. La faisabilité de l'ensemble des chemins frontières ${\varphi_t}_m^k$ détermine si l'injection de la faute d'occurrence k du modèle de faute m sur le chemin de référence φ induit une itération supplémentaire non considérée. Cette itération supplémentaire, non considérée par l'analyse de robustesse, peut potentiellement mener à une vulnérabilité. L'analyse de complétude de l'énumération des chemins fautés considérant un contexte symbolique ctx est définie par la formule F_{compl}.

$$F_{compl} : Fcontexte \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_t \in {\varphi_t}_m^k} Feas(\varphi_t, ctx))$$

où $Fcontexte$ précise des contraintes éventuelles sur les entrées de l'utilisateur appliquées aux variables initiales du chemin de référence (φ_r) et des chemins frontières (φ_t). La formule F_{compl} vérifie si un chemin de référence φ_r et au moins un des chemins frontières φ_t dans ${\varphi_t}_m^k$ est faisable à partir du même contexte symbolique ctx. Nous interprétons le résultat de la résolution de cette formule par un solveur SMT comme suit :

— F_{compl} est satisfiable : au moins une des configurations des entrées autorisées par ctx permet l'exécution d'au moins un chemin fauté (induit par la faute) non considéré par l'analyse de robustesse. Ce chemin pourrait mener au point d'observation de la propriété de sécurité. Pour l'injection du modèle de faute m à l'occurrence k du chemin de référence φ_r, l'énumération des chemins fautés est incomplète.

— F_{compl} est insatisfiable : il n'existe pas de configurations des entrées autorisées par ctx qui permet l'exécution de chemins induits par la faute qui ne sont pas considérés. Pour l'injection du modèle de faute m à l'occurrence k du chemin de référence φ_r, l'énumération des chemins fautés est complète.

4.5.5 Analyse de robustesse

L'analyse de robustesse nécessite de l'utilisateur qu'il spécifie une propriété de sécurité à vérifier au point d'observation.

La propriété de sécurité peut prendre plusieurs formes. Il peut s'agir d'une propriété d'équivalence, vérifiant l'intégrité d'éléments mémorisants d'intérêts au point d'observation. Une telle propriété peut, par exemple, exprimer le fait qu'un registre doit conserver sa valeur avec ou sans la présence d'une faute. Il peut aussi s'agir d'une propriété portant directement sur l'état d'un ou plusieurs éléments mémorisants sans se référer au chemin de référence. Une telle propriété peut, par exemple, exprimer l'égalité du registre `r2` a la valeur du registre `r4` + '2' dans le chemin fauté. Cette propriété est traduite en une formule logique, appelée *Vuln*, qui exprime la négation de la propriété : soit la non-égalité des valeurs d'un sous-ensemble d'éléments mémorisants d'un chemin de référence et d'un chemin fauté, soit la négation de l'expression décrivant l'état des éléments mémorisants au point d'observation du chemin fauté. On considère qu'une faute mène à une vulnérabilité lorsqu'il existe une configuration des entrées permettant la faisabilité du chemin de référence φ_r et d'un chemin fauté φ_f et telle que *Vuln* est vraie.

Étant donné l'ensemble des chemins fautés ${\varphi_f}_m^k$ issu de l'injection du modèle de faute m à l'occurrence k sur le chemin de référence φ_r et pour une configuration des entrées autorisée par le contexte symbolique ctx, la présence d'une vulnérabilité est déterminée par la satisfiabilité de la formule suivante :

$$F_{vuln} : \ Fcontexte \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_f \in {\varphi_f}_m^k} Feas(\varphi_f, ctx)) \wedge Vuln$$

où *Fcontexte* précise des contraintes éventuelles sur les entrées de l'utilisateur appliquées aux variables initiales du chemin de référence (φ_r) et des chemins fautés (φ_f). La formule F_{vuln} vérifie si un chemin de référence φ_r et au moins un des chemins frontières φ_f dans ${\varphi_f}_m^k$ est faisable à partir du même contexte symbolique ctx. Nous interprétons le résultat de la résolution de cette formule par un solveur SMT comme suit :

— F_{vuln} est satisfiable : au moins une configuration des entrées qui est autorisée par ctx permet l'exécution du chemin de référence φ_r et d'un chemin fauté dans ${\varphi_f}_m^k$ telle que *Vuln* est vraie au point d'observation. L'injection du modèle de faute m à l'occurrence k du chemin de référence φ_r met en évidence une vulnérabilité.

— F_{vuln} est insatisfiable : il n'existe pas de configuration des entrées qui est autorisée par

ctx qui permet à la fois l'exécution du chemin de référence φ_r et d'un des chemins fautés dans ${\varphi_f}_m^k$ telle que *Vuln* est vraie au point d'observation. L'injection du modèle de faute m à l'occurrence k sur le chemin de référence φ_r ne met pas en évidence une vulnérabilité.

Certaines contre-mesures reposent sur des mécanismes de détection déroutant l'exécution vers du code de gestion de détection d'erreur. Ces blocs de déroutement, appelés blocs de détection, ne sont pas explicités dans cette analyse. En effet, lorsque les formules F_{compl} et F_{vuln} sont insatisfiables, la faute est considérée comme étant soit tolérée soit détectée. Sinon, lorsque F_{vuln} est satisfiable la faute est considérée comme étant non détectée par un éventuel code réalisant la détection, et ce quelle que soit la satisfiabilité de F_{compl}. Les différents cas de satisfiabilité de F_{compl} et de F_{vuln} sont discutés plus en détails dans la partie suivante.

4.5.6 Complétude

Pour une faute, l'analyse de robustesse est complète si l'énumération des chemins fautés issus de cette faute est complète. L'interprétation des résultats de l'analyse de robustesse pour une faute sur un chemin de référence est discutée ci-dessous en fonction du la satisfiabilité des formules F_{compl} et F_{vuln}.

F_{compl} insatisfiable et F_{vuln} satisfiable : l'analyse est complète et révèle une vulnérabilité. Une nouvelle version du code visant à corriger la vulnérabilité doit être produite et cette nouvelle version du code peut être analysée.

F_{compl} satisfiable et F_{vuln} satisfiable : l'analyse est incomplète et révèle une vulnérabilité. Les chemins fautés non considérés par l'analyse doivent être investigués pour déterminer s'ils peuvent induire une vulnérabilité. Dans tous les cas, le code n'est pas robuste à la faute et une nouvelle version du code visant à corriger la vulnérabilité doit être produite. La nouvelle version du code doit être analysée.

F_{compl} insatisfiable et F_{vuln} insatisfiable : l'analyse ne révèle pas de vulnérabilité et elle est complète.

F_{compl} satisfiable et F_{vuln} insatisfiable : l'analyse ne révèle pas de vulnérabilité mais elle est incomplète. La faute induit des chemins fautés non considérés qui pourraient mettre en évidence une vulnérabilité. L'analyse peut être relancée avec des bornes de boucles encore plus relâchées. Cela pourrait mener à la découverte d'une nouvelle vulnérabilité et/ou éliminer les chemins non explorés. Le résultat d'un relâchement des contraintes sur

les boucles peut donc retomber dans n'importe quel cas parmi les quatre présentés. Le relâchement des bornes de boucles peut augmenter la taille des problèmes SMT jusqu'à atteindre les limites du solveur. Lorsqu'un tel cas survient, une solution externe (*e.g.* revue de code) est nécessaire pour déterminer si les chemins fautés non considérés peuvent mener à une vulnérabilité, et ainsi permettre de peut-être conclure sur la robustesse de la région de code.

Il est important de noter que l'analyse de complétude de l'énumération des chemins fautés n'est pertinente que dans le cas où aucune vulnérabilité n'a été trouvée.

4.6 Exploitation des résultats

Cette partie présente un ensemble de métriques conçues pour aider le concepteur de contre-mesures à gagner en compréhension vis-à-vis des résultats d'une analyse.

Une fois l'analyse réalisée, quand les problèmes SMT associés à toutes les occurrences d'un modèle de faute sur tous les chemins de références sont résolus, une vue globale de la sensibilité de la région de code à ce modèle de faute est représentée au travers d'une liste de vulnérabilités. Le nombre de vulnérabilités trouvées peut être considéré comme un score global de la sensibilité de la région de code vis-à-vis du modèle de faute. Mais cela n'est pas suffisant pour plusieurs raisons. D'abord, toutes les fautes trouvées ne sont pas nécessairement aussi dangereuses : une faute sur une occurrence d'instruction toujours exécutée quel que soit le chemin suivi est plus facile à réaliser que sur une occurrence d'une autre instruction se trouvant sur peu de chemins. De plus, lorsqu'une analyse révèle de multiples vulnérabilités, il peut être difficile de ramener les vulnérabilités au niveau du code assembleur ou encore de déterminer les vulnérabilités affectant la même instruction sur des chemins différents. Il est donc nécessaire d'investiguer l'ensemble des vulnérabilités trouvées pour être en mesure de tirer des conclusions générales sur la vulnérabilité de la région de code.

Pour ces raisons, nous présentons dans la suite trois métriques indiquant à l'utilisateur les instructions les plus sensibles et attribuant des scores globaux à la région de code.

4.6.1 Idée générale : métriques de sensibilité basées sur la probabilité d'exécution des chemins

L'idée commune aux métriques proposées est la suivante : dans le cas général, une vulnérabilité trouvée par l'injection d'une faute sur une instruction se trouvant dans un bloc de base qui a plus chance d'être exécuté est plus dangereuse qu'une autre se trouvant dans un bloc de base ayant moins de chance d'être exécuté.

Ainsi, si une occurrence d'instruction, se situant dans le premier bloc du programme, est sensible sur tous les chemins de référence alors, elle aura plus de poids qu'une autre occurrence d'instruction sensible sur un chemin et se situant dans le dernier bloc de base d'un chemin. Dans le premier cas, la vulnérabilité est liée à une occurrence d'instruction qui sera toujours exécutée quelles que soient les entrées du programme, alors que dans le second cas, la vulnérabilité n'est présente que lorsque la configuration des entrées du programme permet à l'exécution de suivre le chemin faisant apparaître le bloc où se trouve l'occurrence d'instruction vulnérable.

Dans les métriques qui sont présentées dans la suite, nous prenons cela en compte en utilisant la probabilité qu'une exécution emprunte le préfixe de chemin partant de la racine jusqu'à l'occurrence du bloc de base contenant l'instruction vulnérable. La probabilité qu'une exécution emprunte un préfixe de chemin dépend de la probabilité qu'une exécution suive les directions des branchements conditionnels nécessaires à l'apparition de ce préfixe de chemin. Cela implique de connaître la probabilité d'échec (ou de réussite) des branchements conditionnels le long du préfixe de chemin. Cette information n'étant pas nécessairement connue, nous supposons dans la suite que l'échec et la réussite d'un branchement ont tous deux 50 % de chance de se réaliser. Bien que cela ne soit pas représentatif des programmes en général, cela permet tout de même de donner une plus forte pondération aux instructions sensibles apparaissant sur plusieurs chemins qu'à celles n'apparaissant qu'une fois, à la suite de multiples branchements conditionnels. Il est toutefois possible d'ajuster les probabilités d'échec et de réussite des branchements conditionnels si elles sont connues.

Les métriques présentées pondèrent donc chaque vulnérabilité par la probabilité d'exécution de l'occurrence du bloc sur lequel se trouve l'instruction sensible. Cette probabilité est calculée vis-à-vis des chemins faisables par la formule suivante :

$$P(\textit{exécution d'un bloc}) = \left(\frac{1}{2}\right)^{\textit{Nb de branchements conditionnels depuis la racine}}$$

La figure 4.6 montre un CFG et l'arbre des chemins correspondant au déroulage du CFG avec exactement une prise de l'arc arrière $E \rightarrow B$ ainsi que les probabilités associées aux chemins dans le cas où tous les chemins du PEFG sont faisables.

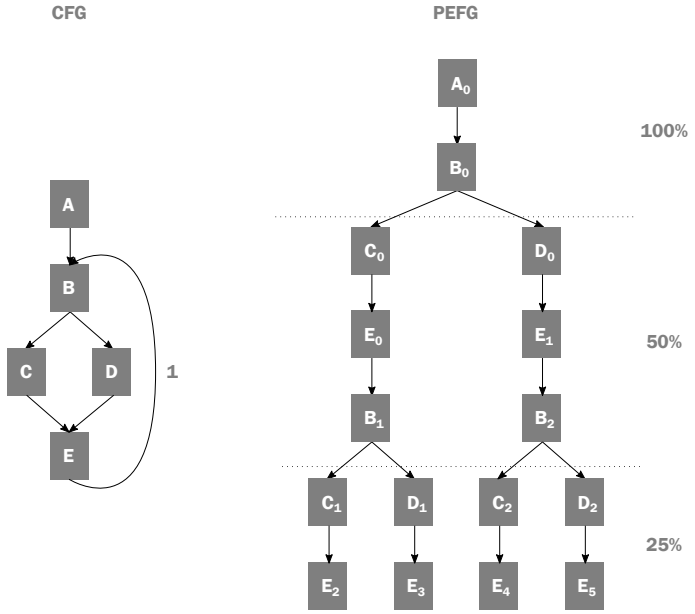

FIGURE 4.6: CFG et probabilités des chemins issus du déroulage du CFG (PEFG) dans le cas où ils sont tous faisables

À chaque branchement conditionnel, la probabilité que l'exécution suive l'un des deux chemins résultant du branchement est deux fois inférieure à celle correspondant au bloc de base dans lequel se situe le branchement.

4.6.2 Sensibilité d'une instruction (IS)

La sensibilité d'une instruction, notée *IS*, est définie pour chaque instruction du programme. Elle représente le nombre moyen de vulnérabilités produites par des fautes affectant les occurrences de l'instruction i. Elle est définie par la formule suivante :

$$IS(i) = \sum_{\varphi \in Chemins} P(\varphi \; est \; emprunté) \times NV_i(\varphi)$$

où $P(\varphi \; est \; emprunté)$ est la probabilité pour un chemin φ d'être emprunté et où $NV_i(\varphi)$ correspond au nombre de vulnérabilités affectant l'instruction i sur le chemin φ. Cette métrique attribue un score à toutes les instructions du programme reflétant leur sensibilité respective, ce qui permet de les ordonner de façon à indiquer quelles sont celles qui doivent être investiguées en priorité.

La figure 4.7 représente un PEFG d'un programme composé de trois chemins valides

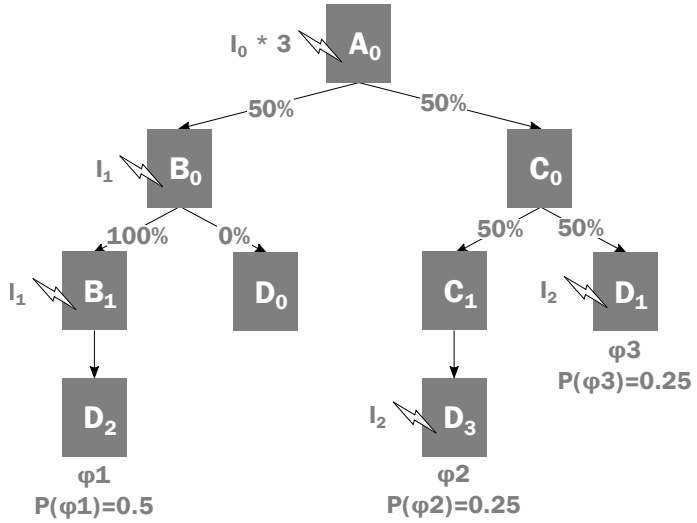

FIGURE 4.7: Exemple pour l'illustration de la sensibilité des instructions. I_0, I_1 et I_2 sont des instructions du code assembleur

($\varphi 1$, $\varphi 2$ et $\varphi 3$) et dans lequel sept vulnérabilités ont été trouvées. L'instruction I_0 est sensible sur les trois chemins du programme, l'instruction I_1 est sensible deux fois sur le chemin $\varphi 1$ et l'instruction I_2 est sensible une fois sur le chemin $\varphi 2$ et une autre fois sur le $\varphi 3$.

Les scores de sensibilité des instructions fautées mettant en évidence une vulnérabilité sont les suivants :

$$IS(I_0) = P(\varphi 1) \times 1 + P(\varphi 2) \times 1 + P(\varphi 3) \times 1 = 1$$

$$IS(I_1) = P(\varphi 1) \times 2 + P(\varphi 2) \times 0 + P(\varphi 3) \times 0 = 1$$

$$IS(I_2) = P(\varphi 1) \times 0 + P(\varphi 2) \times 1 + P(\varphi 3) \times 1 = 0,5$$

Ainsi, comme l'instruction I_1, l'instruction I_0 a une sensibilité de 1 alors que l'instruction I_2 a une sensibilité de 0,5. I_0 est vulnérable à chaque exécution de la région, I_1 a deux occurrences vulnérables sur un chemin ayant 50 % de chance d'être emprunté. L'instruction I_2 est vulnérable sur les chemins $\varphi 2$ et $\varphi 3$ qui ont chacun 25 % de chance d'être exécuté, son score de sensibilité est donc de 0,5.

4.6.3 Surface d'attaque (AS)

La surface d'attaque, notée AS, représente le nombre moyen de vulnérabilités par exécution toutes exécutions confondues. Elle est définie par la formule suivante :

$$AS = \sum_{\varphi \in Chemins} P(\varphi \; est \; emprunté) \times NV(\varphi)$$

où $P(\varphi \; est \; emprunté)$ est la probabilité pour un chemin φ d'être emprunté et $NV(\varphi)$ est le nombre de vulnérabilités présentes sur le chemin φ. La métrique AS donne le nombre moyen de vulnérabilités sur un chemin, elle est donc utile si l'on considère un attaquant qui a la capacité de déclencher la faute avec une bonne précision temporelle mais dont l'effet de la faute n'est pas bien maîtrisé. Cela peut être le cas si la perturbation induite par l'attaque n'a pas toujours d'effet, ou si l'effet n'est pas toujours celui souhaité. Plus AS est élevée plus l'attaquant a de possibilités, par exécution, de réussir son attaque.

FIGURE 4.8: Exemple pour l'illustration de la surface d'attaque

La figure 4.8 illustre la surface d'attaque à travers deux exemples. Le programme A contient 7 vulnérabilités réparties sur deux chemins et le programme B a un unique chemin contenant 4 vulnérabilités. Dans l'exemple A, les vulnérabilités sont présentes sur un chemin ayant une probabilité d'être emprunté de 50 %. Et, le code de l'exemple B n'ayant qu'un seul chemin, les vulnérabilités sont présentes sur un chemin ayant 100 % de chance d'être emprunté. Le calcul de la surface d'attaque des programmes A et B sont :

$$AS(A) = P(\varphi 1) \times 2 + P(\varphi 2) \times 5 = 3.5$$

$$AS(B) = P(\varphi 1) \times 4 = 4$$

Ainsi, le programme A a 7 vulnérabilités mais 3,5 en moyenne par exécution tandis que le programme B a 4 vulnérabilités. Le total de vulnérabilités est plus élevé dans le programme A que dans le programme B mais la surface d'attaque du programme B est supérieure à celle du programme A.

4.6.4 Surface d'attaque normalisée (NAS)

La surface d'attaque normalisée (notée NAS) indique la probabilité moyenne, par chemin, qu'une faute résulte en une vulnérabilité. Elle est donnée par la formule suivante :

$$NAS = \frac{\sum_{\varphi \in Chemins} P(\varphi \ est \ emprunté) \times NV(c)}{\sum_{\varphi \in Chemins} P(\varphi \ est \ emprunté) \times NI(\varphi)} = \frac{AS}{ANI}$$

où $NI(\varphi)$ est le nombre de points d'injection (cf. section 4.4.2) le long du chemin φ. ANI représente le nombre moyen de points d'injection par chemin. Cette métrique est utile si l'attaquant considéré ne contrôle pas le moment exact où l'injection est réalisée mais aussi lorsque l'attaquant considéré n'a pas connaissance du code et réalise donc des injections aléatoirement pendant l'exécution de la région de code.

FIGURE 4.9: Exemple pour l'illustration de la surface d'attaque normalisée

La figure 4.8 illustre la surface d'attaque normalisée avec deux exemples ayant chacun deux chemins avec la même probabilité d'être emprunté et quatre vulnérabilités reparties de la même façon. Toutefois, chaque bloc du programme de l'exemple A contient 100 instructions quand dans l'exemple B ils en contiennent 10. Considérant le modèle de saut d'instruction, le calcul de la surface d'attaque normalisée pour les programmes A et B sont les suivants :

$$NAS = \frac{P(\varphi 1) * 3 + P(\varphi 2) * 3}{P(\varphi 1) * 200 + P(\varphi 2) * 200} = 0.1$$

$$NAS = \frac{P(\varphi 1) * 3 + P(\varphi 2) * 3}{P(\varphi 1) * 20 + P(\varphi 2) * 20} = 0.01$$

Ainsi, une attaque aléatoire pendant l'exécution de la région de code du programme A a 10 % de chance de réussir contre 1 % de chance pour le programme B.

4.7 Implémentation

Dans cette partie, nous présentons `RobustB`, l'outil qui implémente la méthode présentée précédemment. `RobustB` utilise quelques outils existants mais son cœur a été développé pendant cette thèse.

4.7.1 RobustB

Les entrées de `RobustB` sont : le programme binaire incluant la région de code à analyser, une propriété de sécurité à vérifier à la fin de la région de code portant sur des éléments mémorisants et un modèle de faute. `RobustB` détermine la robustesse de la région de code par l'application d'un modèle de faute à tous les points d'injection possibles sur tous les chemins possibles de la région de code. À l'issue de l'analyse, les résultats des différentes injections de faute sont recueillis pour produire la liste de vulnérabilités et calculer les métriques, à savoir : la sensibilité des instructions, la surface d'attaque et la surface d'attaque normalisée.

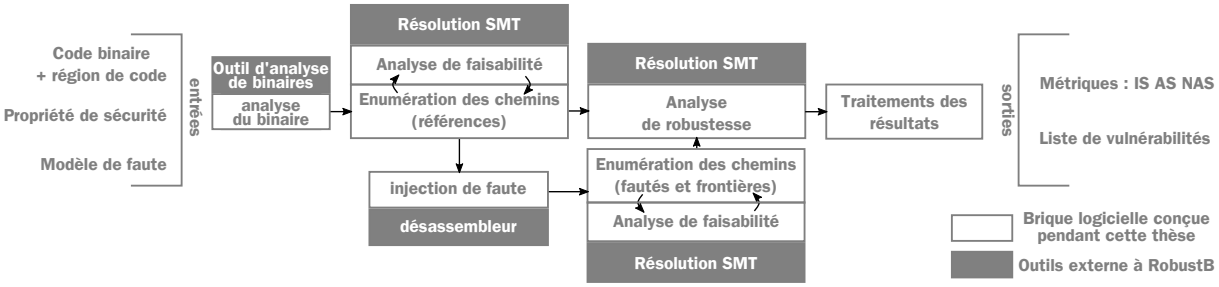

FIGURE 4.10: Vue schématique de `RobustB`

La figure 4.10 présente les principaux modules de `RobustB`, à savoir : l'analyse du programme binaire, l'énumération des chemins, l'injection de faute, la construction des problèmes SMT de faisabilité et de robustesse, la résolution des problèmes SMT et le traitements des résultats. `RobustB` extrait d'abord des informations sur la structure et le contexte symbolique de la région de code à analyser. De ces informations, `RobustB` construit l'ensemble des chemins de références en utilisant une analyse statique et une analyse de faisabilité faisant appel à un solveur SMT. L'injection de faute applique les transformations associées au modèle de faute à tous les points d'injection possibles le long de chaque chemin de référence. Une nouvelle énumération de chemins, extrayant les chemins fautés et les éventuels chemins frontières, est réalisée pour chaque faute injectée. Une analyse de robustesse est ensuite réalisée à partir d'un chemin fauté et du chemin de référence dont il est issu. Enfin, les résultats des résolutions SMT sont rassemblés pour former le résultat de l'analyse : la liste des vulnérabilités et les valeurs des trois métriques.

4.7.1.1 Hypothèses sur le code analysé

Les programmes analysés doivent respecter un certain nombre de contraintes pour que l'analyse puisse être réalisée et que ses résultats soient exploitables.

— Si la région analysée contient des appels de fonctions alors le code de ces fonctions doit se trouver dans le binaire (cela est vrai récursivement). Autrement dit, l'ensemble du code exécutable à partir de la région de code doit se trouver dans le binaire.

— Les interruptions et exceptions ne sont pas gérées lors de l'analyse formelle. Elles peuvent toutefois être simulées lors de l'analyse du programme binaire.

— L'analyse considère un seul point d'observation de la région de code, point auquel la propriété est vérifiée.

En ce qui concerne le dernier point, jusque-là nous avons toujours considéré que le point de sortie et le point d'observation étaient confondus. Or, lorsque la région de code est une fonction, *i.e.* le point d'observation se trouve dans la fonction, il se peut que celle-ci ait deux sorties ou plus. Prenons l'exemple d'une fonction réalisant la vérification d'un code PIN saisi par l'utilisateur. Une telle fonction peut avoir deux sorties (identifiées par l'instruction de dépilage `pop`) : une première sortie dans le cas où le code PIN saisi par l'utilisateur est faux et une seconde sortie dans le cas où le code PIN saisi par l'utilisateur est le bon. Si l'on souhaite réaliser une analyse répondant à la question "existe-t-il une faute qui permet d'atteindre la seconde sortie alors que le code PIN saisi par l'utilisateur est faux ?" alors les chemins de référence (non fautés) termineront tous dans la première sortie et l'analyse cherchera une faute qui permet d'atteindre la seconde sortie. Dans ce cas particulier, les chemins de référence sont déroulés en fonction de la première sortie et les chemins fautés sont déroulés en fonction de la seconde. Le point d'observation, où la propriété est à vérifier correspond alors à la seconde sortie. Une façon d'appréhender la résolution de ce problème peut consister en la création d'un nouveau bloc dans le CFG auquel les deux sorties sont reliées.

4.7.1.2 Interactions

Dans cette partie, nous présentons les différents modules de `RobustB` et leurs interactions. La figure 4.11 présente une vue schématique des interactions entre les principaux modules du cœur de `RobustB` lors de l'injection d'une seule faute de type instruction bit-set ou instruction bit-reset pour un contexte symbolique donné. Les colonnes représentent les différents modules de `RobustB`, présentés dans la suite, et les flèches indiquent ce qui

est échangé entre les modules. Nous décrivons ensuite les interactions de chaque module séparément.

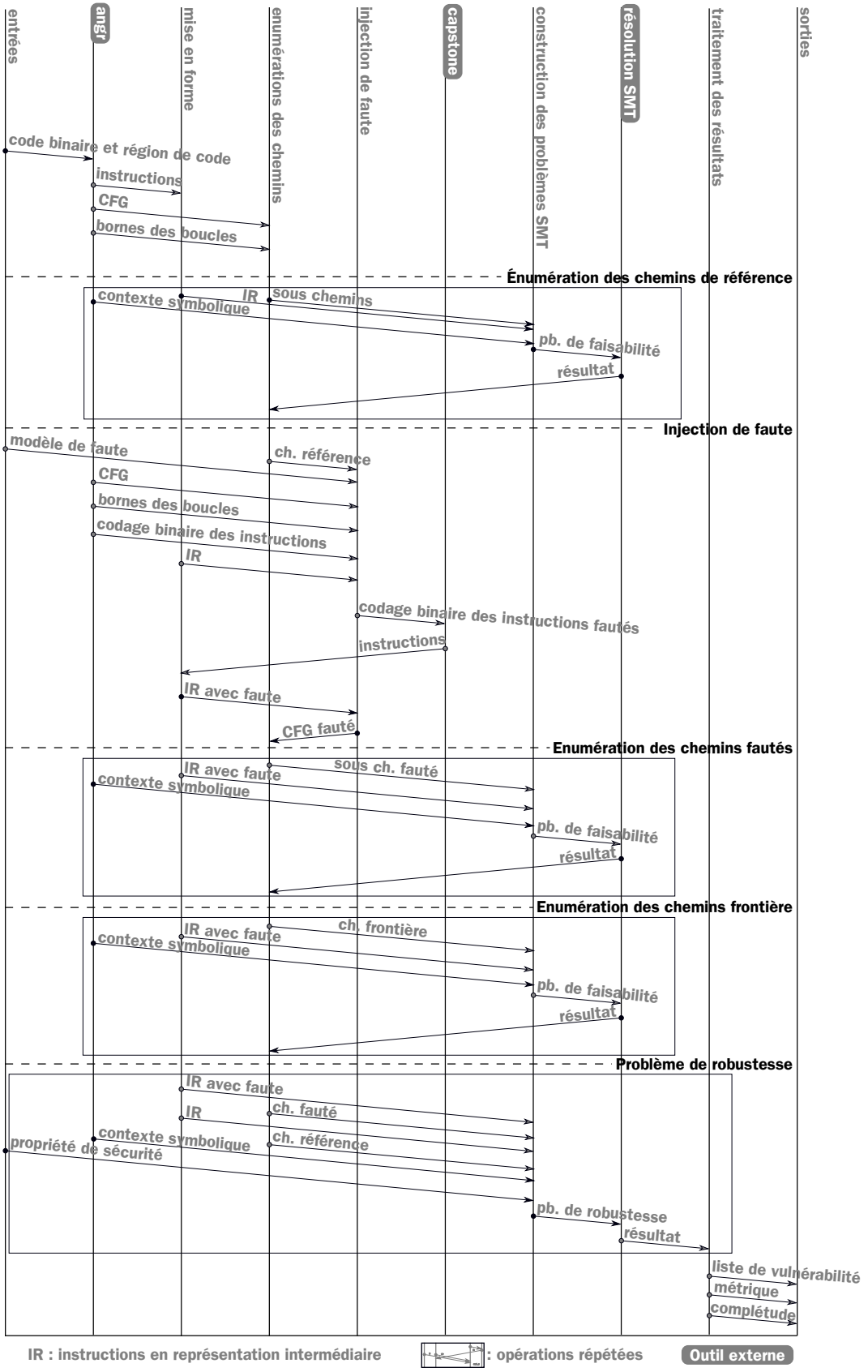

FIGURE 4.11: Diagramme des interactions de `RobustB` et de ses outils

Infrastructure `angr` [SWS+16]. C'est une infrastructure écrite en python permettant l'analyse de codes binaires. `angr` est capable d'effectuer plusieurs types d'analyses statiques et dynamiques (moteur d'exécution symbolique). Il est utilisé par `RobustB` pour l'extraction de différentes informations du binaire. Il prend en entrée le fichier binaire, au format ARM Thumb-2, ainsi que l'indication concernant la région de code. La région de code peut être indiquée soit par un nom de fonction (lorsque l'analyse porte sur une fonction), soit par deux adresses correspondant au point d'entrée et au point de sortie de la région de code à analyser. À partir de ces données, les analyses réalisées avec l'aide d'`angr` produisent cinq éléments :

— le graphe de flot de contrôle (CFG) de la région de code à analyser
— les instructions en assembleur ARM Thumb-2 avec leur adresse reflétant leur placement dans le CFG
— les contextes symboliques, *i.e.* les différents ensembles d'expressions symboliques des éléments mémorisants pertinents pour la région de code à analyser
— les bornes des boucles de la région de code à analyser pour chaque contexte symbolique
— le codage binaire de la région de code à analyser

Mise en forme. Ce module traduit les instructions assembleur en une représentation intermédiaire *ad-hoc*. Cette représentation intermédiaire décompose chaque instruction en plusieurs opérations de façon à ce que les instructions soient plus facilement exprimées avec les opérations offertes par les théories SMT. À la suite de ce formatage, on associe chaque bloc de base du CFG à un ensemble d'instructions exprimées dans cette représentation intermédiaire. Ce module traduit aussi les contextes symboliques, issus de l'analyse du binaire, pour qu'ils puissent être donnés en entrée au solveur SMT.

Énumération des chemins. Ce module a pour but de déterminer l'ensemble des chemins, menant au point d'observation à partir du CFG, d'un contexte symbolique et des éventuelles bornes des boucles associées à ce contexte. Le CFG est déroulé progressivement, et lors de ce déroulage, les problèmes SMT de faisabilité des chemins intermédiaires (sous-chemins) sont générés, puis résolus par un solveur SMT. Au terme de son exécution, ce module produit un ensemble de chemins faisables du CFG pour le contexte symbolique donné et les bornes de boucles associées. Le module peut dérouler deux types de CFG : un CFG original (issu de l'analyse du programme binaire) et un CFG fauté (issu d'une injection de faute). Les chemins faisables issus du déroulage d'un CFG original sont des chemins de références et ceux issus d'un CFG fauté sont des chemins fautés. Dans le cas d'un déroulage d'un CFG fauté, l'énumération des chemins calcule aussi l'ensemble des chemins frontières.

Injection de faute. Ce module agit sur la représentation intermédiaire d'un chemin de référence pour y appliquer la transformation associée à un modèle de faute. Nous avons déjà vu qu'une faute peut affecter le flot de contrôle et qu'il est donc nécessaire de déterminer les chemins faisables après la faute. Ce module prend donc aussi en compte les éventuelles modifications du flot de contrôle induites par la faute sur un chemin de référence. Certains modèles de faute (*e.g.* instruction bit-set) modifient le codage binaire des instructions. Dans ce cas, le nouveau codage binaire est transmis à `capstone` [Quy14], un outil de désassemblage, qui interprète le codage binaire des instructions, par exemple lors d'une faute de type instruction bit-reset, et renvoie les instructions assembleur correspondantes.

Construction des problèmes SMT. Ce module construit les problèmes SMT. Les problèmes SMT de faisabilité sont construits à partir d'une séquence d'instructions exprimées dans la représentation intermédiaire et d'un contexte symbolique. Les problèmes SMT de robustesse sont, quant à eux, construits à partir d'une séquence d'instructions issue d'un chemin fauté, de la séquence d'instructions du chemin de référence sur lequel a été injectée la faute, de la propriété de sécurité et d'un contexte symbolique. Ces problèmes sont ensuite donnés à un solveur SMT et le résultat de la résolution est recueilli.

Traitements des résultats. Ce module analyse les résultats des différentes résolutions des problèmes SMT, génère une liste des vulnérabilités et calcule les valeurs des 3 métriques. Les paramètres correspondant à la probabilité qu'un chemin soit emprunté et le nombre de points d'injection le long des chemins sont calculés par `RobustB` pendant l'analyse. Tous les autres paramètres peuvent être déduits des informations présentes dans la liste des vulnérabilités.

4.7.1.3 Analyse du programme binaire

L'infrastructure `angr` est une boîte à outils écrite en python proposant diverses analyses sur du code binaire ARM Thumb-2. La plus grande partie des travaux relatifs à l'analyse du programme binaire sont issus du stage de fin d'étude de Son Tuan Vu [Vu17].

L'outil `angr` prend en entrée le programme binaire et deux informations : l'adresse du point d'entrée et celle du point de sortie de la région de code à analyser. Lors de l'analyse en charge d'extraire les contextes symboliques en entrée de la région de code, le moteur d'exécution symbolique d'`angr` exécute le programme jusqu'à atteindre la première instruction de la région de code. Une stratégie naïve part du début du programme et explore les différents chemins en implémentant un parcours en largeur ou en profondeur. Ces stratégies sont inadaptées à de gros programmes car non dirigées, elles n'ont d'autres choix que d'explorer tous les chemins possibles et sont donc lentes et potentiellement

consommatrices en mémoire. Nous avons opté pour une solution implémentant d'abord une coloration de graphe dans le CFG du programme complet. L'algorithme de coloration de graphe (inspiré des travaux de Potet *et al.* [PMPD14]) part du bloc de base du point d'entrée de la région de code et "remonte" dans le CFG jusqu'au point d'entrée du programme en marquant les blocs sur son passage. L'exécution symbolique peut, de cette façon, être dirigée : elle n'explore plus que les chemins contenant des blocs susceptibles de mener à un point d'entrée de la région de code. Aussi, nous avons désactivé les stratégies d'approximation d'*angr* au prix d'une potentielle non réponse de sa part. Cela a pour conséquence que si la coloration de graphe à effectivement découvert toutes les séquences de blocs pouvant mener à la région de code alors l'exécution symbolique faite par *angr* est complète.

4.7.1.4 Énumération des chemins

L'énumération des chemins peut être décomposée en deux étapes : la construction du PEFG et l'analyse de faisabilité des sous-chemins du PEFG. La première étape consiste en le déroulage statique du CFG pour former le PEFG. Lors de ce déroulage, les deux directions possibles du branchement conditionnel sont considérées (par exemple, les branches *then* et *else* d'un *if* sont toutes deux prises) sauf lorsque l'occurrence de l'instruction de branchement conditionnel correspond à une sortie de boucle. La figure 4.12 montre un exemple de déroulage d'un CFG contenant un test dans une boucle dont l'arc arrière est pris au plus une fois. Le PEFG résultant contient six chemins potentiels, deux sans prise de l'arc arrière $E \rightarrow B$ et quatre où cet arc a été pris une fois.

Une fois le PEFG obtenu, la faisabilité des chemins est analysée. La façon naïve de procéder est d'analyser la faisabilité de chacun des chemins du PEFG. Une stratégie naïve pour déterminer l'ensemble des chemins possibles de la région de code est de construire le PEFG puis de vérifier la faisabilité de ses chemins un à un. Toutefois, la forme de certains programmes induit un grand nombre de chemins potentiels alors que très peu d'entre eux sont faisables pour le contexte symbolique donné. Par exemple, le nombre de chemins potentiels d'un programme ayant une boucle contenant un `if` croît exponentiellement avec le nombre de tours de boucle considéré. La figure 4.13 montre le PEFG résultant du déroulage d'un CFG affichant une telle complexité ($\sim$2200 chemins). Ce CFG représente une fonction réalisant une authentification par code PIN protégée à la compilation contre un saut d'instruction [Mor14, BCR16]. L'approche choisie pour accélérer la découverte

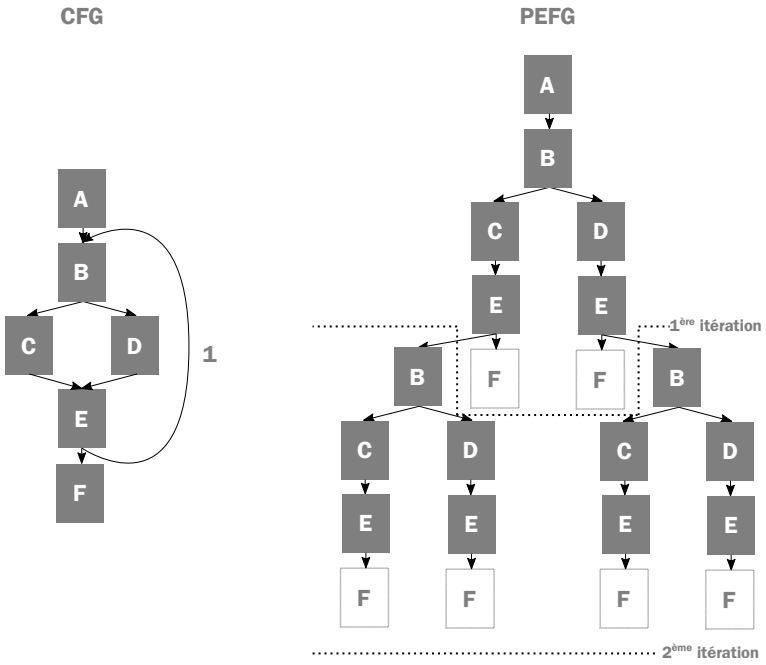

FIGURE 4.12: Exemple de génération d'un PEFG

des chemins faisables est la même que celle réalisée lors d'une exécution symbolique qui s'apparente une découverte incrémentale des chemins et sous-chemins faisables. La stratégie employée construit le PEFG puis teste la faisabilité des sous-chemins du PEFG pas à pas depuis sa racine. Lorsque l'exploration rencontre un bloc ayant deux arcs sortants menant à deux blocs, B_A et B_B, la faisabilité des chemins partant du bloc racine jusqu'aux blocs B_A et B_B est modélisée puis vérifiée formellement. La faisabilité des chemins contenus dans les sous arbres ayant comme racine B_A et B_B n'est réalisée que s'ils sont respectivement faisables. Cette stratégie permet de réduire considérablement la durée de l'énumération des chemins de références. La stratégie optimale devrait tester la faisabilité des sous-chemins pendant la construction du PEFG mais celle-ci n'a pas été réalisée : le gain de performance par rapport à celle implémentée est négligeable tant le temps d'exécution nécessaire à la construction du PEFG est petit devant le temps de résolution des problèmes SMT.

La figure 4.14 montre des exemples d'analyses incrémentales de faisabilité réalisées sur un PEFG. Le PEFG est exploré depuis sa racine et, à chaque branchement rencontré, les problèmes de faisabilité des deux chemins issus du branchement sont créés et résolus grâce à un solveur SMT. L'exploration se poursuit uniquement à partir des séquences de blocs de base pour lesquelles le solveur a déterminé qu'elles étaient faisables (SAT). Les problèmes de faisabilité sont résolus parallèlement par des instances différentes du solveur SMT.

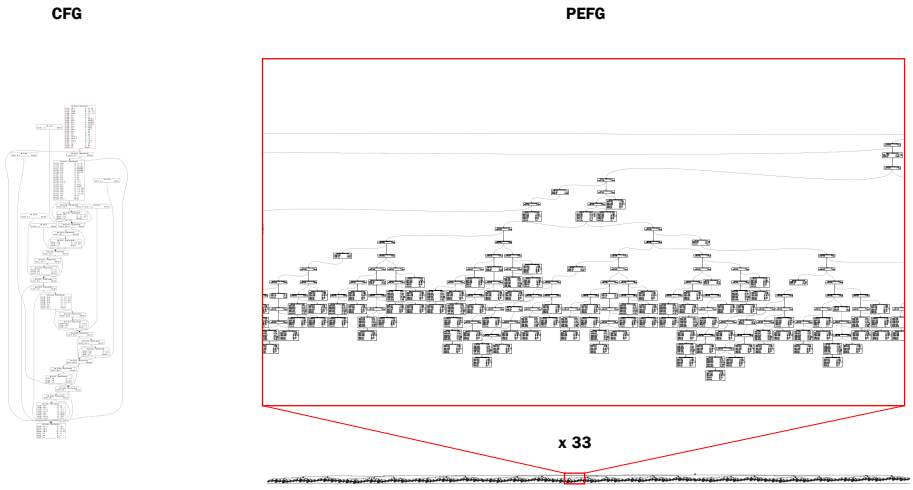

FIGURE 4.13: Exemple de CFG et du PEFG associé d'un code

4.7.1.5 Injection de faute

Le module d'injection de faute interprète l'effet de la faute sur le flot de contrôle et le répercute sur le CFG. Lorsque le modèle de faute choisi est la mise à 1 d'un bit dans l'encodage d'une instruction (instruction bit-set), ce module fait appel à un désassembleur pour réinterpréter l'encodage des instructions. Ce module prend aussi en compte un cas particulier où le compilateur entrelace l'implantation des données et des instructions. Les données, se trouvant entre deux instructions d'une même fonction, peuvent ainsi être exécutées suite à l'injection d'un saut d'instruction sur l'instruction placée juste avant les données. En effet, en ARM Thumb-2, le compilateur peut placer des données, qui correspondent à des adresses d'autres données, entre deux instructions d'une fonction. Cela lui permet de lire cette adresse en utilisant une instruction de lecture relative au compteur ordinal (PC) encodée sur 16 bits plutôt que d'utiliser deux instructions pour charger l'adresse dans un registre. Cette optimisation permet de réduire la taille du code. Un tel bloc de données est souvent placé à la fin du code des fonctions lorsqu'elles contiennent peu de code. Mais, pour les fonctions plus longues, le nombre de bits d'*offset* disponible dans les instructions de lecture mémoire relative peut ne pas être suffisant pour atteindre la fin de la fonction depuis l'instruction de lecture, ce qui est résolu par le compilateur en plaçant le bloc de données au milieu des instructions de la fonction.

La figure 4.15 illustre un cas de l'application du modèle de faute de saut d'instruction donnant lieu à l'exécution de données. On suppose un code composé de 6 blocs de base nommés A, B, C, D, E et F et où des données ont été placées en séquence du bloc de base C. On considère le cas où une faute de type saut d'instruction est injectée sur la dernière instruction du bloc C. Cette faute force l'exécution à continuer en séquence, c'est-à-dire à ce que le processeur interprète les données comme des instructions et les exécute. Le

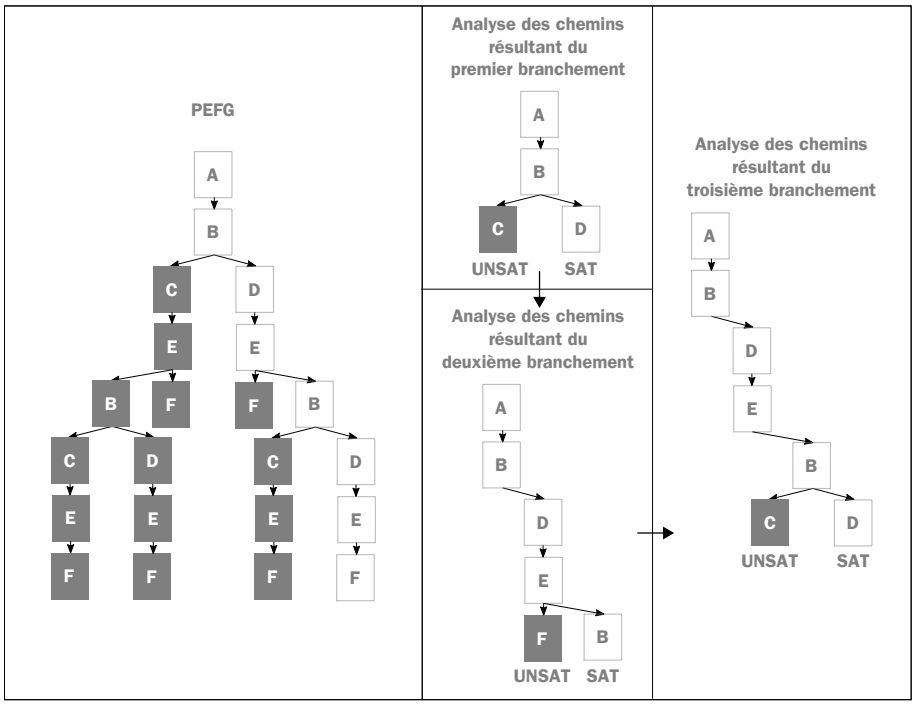

FIGURE 4.14: Exemple pour l'illustration de la recherche incrémentale de chemins dans le PEFG

processus de construction du CFG fauté est réalisé comme suit :

1. Placement des blocs de base dans le programme binaire. Les données sont situées sous le bloc de base C.
2. L'encodage binaire des données est donné au désassembleur et le bloc d'instructions en résultant est inséré dans le CFG initial.
3. Le module d'injection de faute reçoit un chemin de référence et le modèle de faute de saut d'instruction à appliquer à la dernière instruction du bloc 'C' (*i.e.* une instruction de branchement inconditionnel vers le bloc 'E').
4. Le chemin de référence est sectionné à partir de la faute puis fusionné avec le CFG incluant les données vues comme des instructions. Ici, le saut de la dernière instruction du bloc 'C' revient à ignorer le branchement inconditionnel et continuer l'exécution en séquence, soit, à exécuter le bloc de données.

Tous les modèles de faute sont appliqués à tous leurs points d'injection excepté pour le modèle de faute de corruption de registre. En effet, le modèle de faute de corruption de registre n'est appliqué à un registre que lorsque celui-ci est utilisé dans l'instruction suivante dans le chemin. De fait, deux corruptions d'un même registre, réalisées entre deux de ces lectures successives, produisent le même résultat. Cette optimisation réduit le

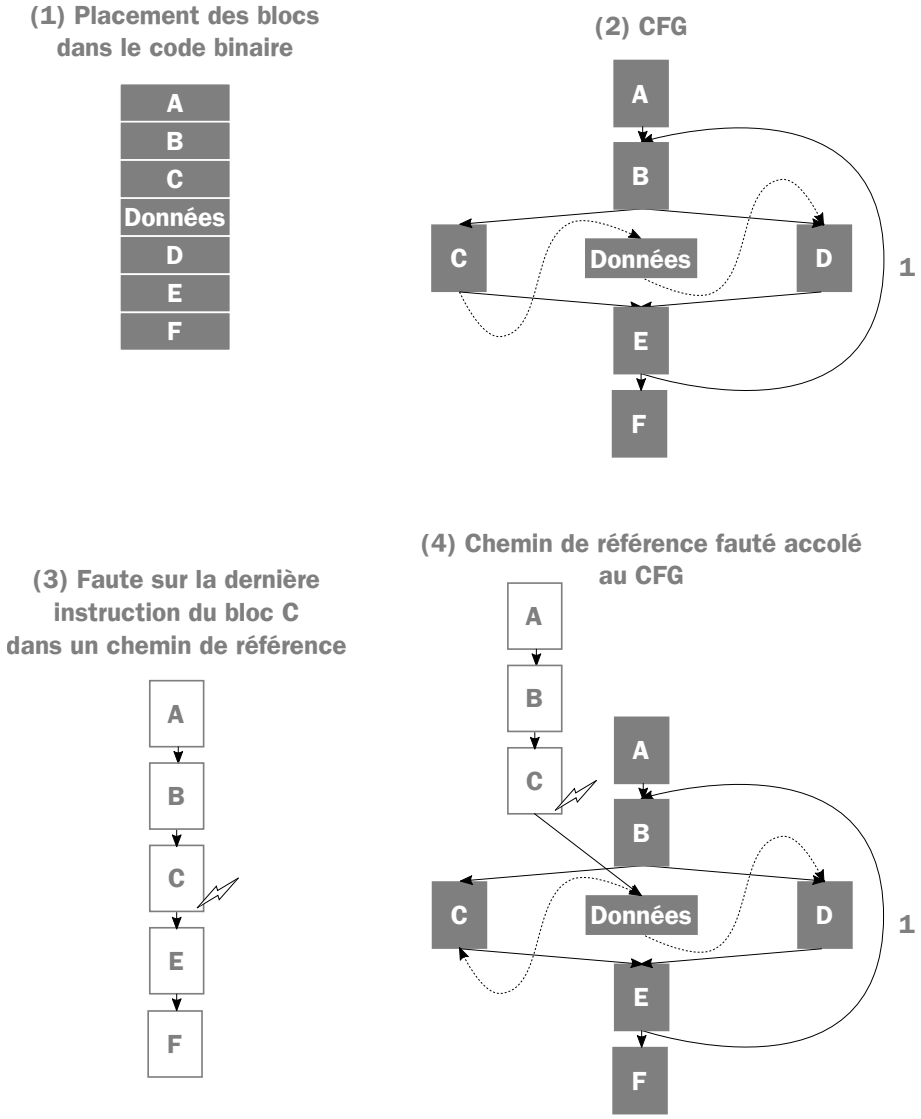

FIGURE 4.15: Impact de l'injection d'une faute induisant l'exécution de données

nombre d'injections de fautes effectivement réalisées. Lorsqu'une corruption de registre met en évidence une vulnérabilité, `RobustB`, pour l'établissement du score des métriques, considère que cette même vulnérabilité existe sur toutes les instructions précédentes jusqu'à ce qu'une instruction lisant ou écrivant le registre soit rencontrée.

4.7.1.6 Construction des problèmes SMT

Le module de construction des problèmes SMT traduit le contexte symbolique et la représentation intermédiaire associée à un chemin en langage SMT. La modélisation des problèmes SMT utilise deux théories : *bit-vectors* et *arrays*. La théorie des bit-vectors permet la déclaration de variables contenant un nombre arbitraire de bits et offre des opérations arithmétiques et logiques ainsi que l'extraction ou encore la concaténation. Elle

est aussi utilisée pour représenter les éléments mémorisants (registres et cases mémoire). La théorie des arrays permet la déclaration de *maps* (en français, tables associatives ou dictionnaires) qui offrent des opérations de lecture et écriture sur ses éléments. Les éléments des maps peuvent être de n'importe quel type et peuvent être indexés par n'importe quel type. Dans notre cas, les éléments des maps sont des bit-vectors de 32 bits et sont utilisés pour représenter une case mémoire. Ces éléments sont indexés par des bit-vectors de 30 bits permettant d'adresser 4 octets alignés en mémoire. La combinaison de ces deux théories est souvent utilisée dans le cadre de la vérification de programme [NPB15, GD07, FBBP18]. Elle permet une représentation naturelle des éléments mémorisants et de la mémoire.

Les problèmes SMT sont générés au format standard SMT-LIB [BST+10]. SMT-LIB a été proposé en 2003 dans le but d'unifier les interfaces avec les différents solveurs SMT existant. Ce standard est adopté, au moins partiellement, par un nombre croissant de solveurs, Z3 [DMB08], Boolector [NPB15], CVC4 [BCD+11], ce qui permet à l'utilisateur de choisir le solveur SMT. Toutefois, des tests sur les cas d'étude que nous avions à notre disposition ont déterminé que le solveur Z3 est plus performant considérant notre modélisation du problème.

Dans le cas de la construction d'un problème de robustesse, la propriété peut être directement exprimée en SMT-LIB. Portant sur des éléments mémorisants et donc sur des bit-vectors, elle est exprimée en utilisant les opérations associées aux bit-vectors offertes par la théorie dans le standard SMT-LIB. Exprimer la propriété en SMT-LIB requiert de l'utilisateur qu'il fasse le lien entre les registres dans le code assembleur et les variables à plus haut niveau. Dans [VHdGC20], Vu *et al.* proposent de propager des informations depuis le code jusqu'au binaire pour, notamment, faciliter l'expression des propriétés. Les travaux des auteurs permettent la propagation d'annotations depuis un code source C jusqu'au code binaire, tout en garantissant la non-optimisation, par le compilateur, des variables utilisées dans l'annotation. Ces travaux permettent l'expression des propriétés au niveau du code C et donnent, après compilation, l'expression des propriétés au niveau binaire. À l'issue de la compilation, les annotations peuvent être retrouvées dans la section *.debug* du fichier binaire au format *DWARF* [5]. Avec l'aide des auteurs, nous avons implémenté la lecture de la propriété depuis le DWARF et sa traduction en SMT-LIB. Les propriétés peuvent être exprimées en utilisant les opérateurs arithmétiques et logiques basiques en plus des opérateurs de déréférencement du langage C. Il est aussi possible, en plus de la propriété, d'indiquer le point d'observation au niveau du code source. Avec la possibilité d'exprimer ces deux informations au niveau du code source, l'analyse de robustesse est complètement automatisée pour des cas simples. De plus, l'utilisateur n'interagit plus du tout au niveau SMT et n'a donc pas nécessairement besoin de compétences à ce niveau pour réaliser

5. DWARF Debugging Standard : http ://dwarfstd.org/

l'analyse de robustesse. L'annexe B montre un exemple de passage de propriété depuis le code source.

4.7.2 Opérations sur les contextes symboliques

Les contextes symboliques sont extraits par les exécutions symboliques réalisées à l'aide de `angr`. Dans le but de réaliser des analyses plus spécifiques, `RobustB` permet d'ajouter de nouvelles contraintes en entrée de la région de code de façon à préciser l'intervalle de définition d'une variable ou d'ajouter des relations entre des variables.

Reprenons l'exemple vu plus tôt (cf. 4.7.1.1) d'une fonction réalisant la vérification de code PIN saisi par l'utilisateur contenant deux sorties et d'une analyse visant à répondre à la même question "existe-t-il une faute qui permet d'atteindre la seconde sortie (associée à une authentification) alors que le code PIN saisi par l'utilisateur est faux ?". Pour pouvoir répondre à la question, il faut pouvoir indiquer que les deux codes PIN doivent être différents. Dans ce cas, il est possible de renseigner une nouvelle relation précisant la différence des deux codes PIN. Ces contraintes additionnelles peuvent être ajoutées directement en SMT-LIB ou données sous forme d'annotations au niveau source (cf. section 4.7.1.6). Ces contraintes additionnelles correspondent au prédicat *Fcontexte* dans la formule F_{vuln} (cf. 4.5.5).

`RobustB` offre aussi la possibilité de symboliser une variable globale dans le binaire. Cette fonctionnalité est utile si, par exemple, le code PIN attendu est initialisé durant l'exécution du programme et qu'il n'est pas possible de recompiler le programme dans le but de supprimer ces affectations. Dans ce cas, les contraintes additionnelles ne suffisent pas car l'exécution symbolique, lors de l'analyse du programme binaire, peut produire des contextes symboliques dans lesquels certaines expressions dépendent de la valeur du code PIN attendu. La symbolisation d'une variable doit donc être réalisée lors de l'exécution symbolique. Les variables à symboliser sont renseignées et, lors de l'exécution symbolique, un mécanisme de fonction de rappel (*callback*) interrompant l'exécution, permet de détecter et de supprimer les écritures réalisées sur les variables renseignées.

4.7.3 Illustration d'une analyse de robustesse

Dans cette partie, nous illustrons les éléments principaux présentés dans ce chapitre sur un code assembleur. Nous définissons d'abord le code assembleur utilisé puis un contexte symbolique (configuration initiale des registres et cases mémoire). Nous définissons ensuite

la propriété ainsi que le modèle de faute choisi. Enfin, nous expliquons les analyses réalisées, leurs résultats, pour conclure en donnant le score de chaque métrique.

4.7.3.1 Éléments en entrée de l'analyse

Le listing 4.2 montre un code assembleur qui assigne la valeur 0 ou 2 au registre `r2` (respectivement @c et @e) en fonction de la valeur du registre `r0` (@a et @b) puis, incrémente le registre `r3` avec la valeur du `r2` (@f). L'instruction `cmp r0, #0` compare le contenu du registre `r0` avec la valeur 0 et met à jour des drapeaux (*flags*) indiquant le résultat de la comparaison. Seul le drapeau nommé `Z` nous intéresse ici, c'est la valeur de ce drapeau qui conditionne la prise ou non du branchement à l'instruction @b. Le drapeau `Z` est mis à 1 (silencieusement) par l'instruction `cmp` lorsque le résultat de la comparaison est égal à 0 et à 0 sinon. L'instruction @a correspond au point d'entrée de la région de code tandis que l'instruction @f correspond au point d'observation.

```
1 @a: cmp     r0, #0    % comparaison de r0 avec #0
2 @b: beq     @e        % si r0 == 0; alors saut à @e
3 @c: movs    r2, #0    % r2 <- 0
4 @d: b       @f        % saut à @f
5 @e: movs    r2, #2    % r2 <- 2
6 @f: add     r3, r2    % r3 <- r3 + r2
```

Listing 4.2: Code assembleur pour l'illustration de la méthode

Le CFG et le placement des blocs associés au code assembleur sont montrés dans la figure 4.16.

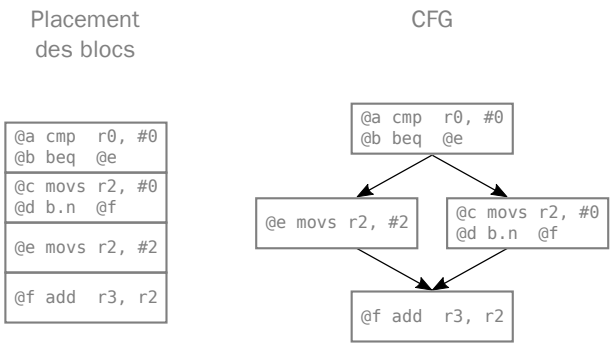

FIGURE 4.16: Placement des blocs et CFG du code assembleur

Le contexte symbolique choisi pour cet exemple est le suivant :

— `r0` = 0x00000000

— `r2` = 0x00000000

— `r3` = 0x00000003

— drapeau `Z` = X

Comme nous pouvons le voir, la valeur du registre `r0` est initialisée à 0 ce qui a pour effet de forcer l'exécution, sans faute, à prendre le branchement à @b et donc d'assigner la valeur 2 au registre `r2` puis d'ajouter la valeur de registre au contenu du registre `r3`. Puisque ce dernier est initialisé à la valeur 3 dans le contexte symbolique, son contenu final vaut 5. La valeur X initialisant le drapeau `Z` indique que la variable est libre et que le choix de la valeur du drapeau `Z` est donc initialement laissé au solveur SMT.

Le modèle de faute choisi, appliqué à tous ses points d'injection, est le modèle de faute de saut d'instruction.

Deux propriétés sont possibles :

$$prop(équivalence) : r3_{V_f}^{\varphi r} = r3_{V_f}^{\varphi f} \tag{4.1}$$

$$prop(spécifique) : r3_{V_f}^{\varphi f} = 0x00000005 \tag{4.2}$$

avec x_z^y désignant la variable x du chemin y de l'ensemble z des éléments mémorisants. L'ensemble V_f désigne l'ensemble des éléments mémorisants à la suite de l'application des transformations de la dernière instruction (@f). La propriété d'équivalence compare la valeur du registre `r3` à la fin du chemin de référence avec celle du même registre à la fin du chemin fauté tandis que la propriété spécifique compare la valeur du registre `r3` à la fin du chemin fauté avec la valeur attendue (5).

Pour cet exemple nous utilisons la propriété d'équivalence. La formule *Vuln* de la formule F_{vuln} est définie comme suit :

$$prop : r3_{V_f}^{\varphi r} = r3_{V_f}^{\varphi f} \tag{4.3}$$

$$Vuln = \overline{prop} \tag{4.4}$$

$$Vuln : r3_{V_f}^{\varphi r} \neq r3_{V_f}^{\varphi f} \tag{4.5}$$

La formule F_{vuln}, utilisée pour l'analyse de robustesse, devient donc :

$$F_{vuln} : \ Fcontexte \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_f \in {\varphi_f}_m^k} Feas(\varphi_f, ctx)) \wedge (r3_{V_f}^{\varphi r} \neq r3_{V_f}^{\varphi f})$$

avec $Fcontexte = \varnothing$ et avec *ctx* désignant la formule initialisant les variables dans V_0

avec le contenu du contexte symbolique dans le chemin de référence et dans le chemin fauté ; elles sont définies comme :

$$\texttt{r0} : 0x00000000 = r0_{V_0}^{\varphi r} = r0_{V_0}^{\varphi f} \tag{4.6}$$

$$\texttt{r2} : 0x00000000 = r2_{V_0}^{\varphi r} = r2_{V_0}^{\varphi f} \tag{4.7}$$

$$\texttt{r3} : 0x00000003 = r3_{V_0}^{\varphi r} = r3_{V_0}^{\varphi f} \tag{4.8}$$

$$\textit{drapeau } \texttt{Z} : X = Z_{V_0}^{\varphi r} = Z_{V_0}^{\varphi f} \tag{4.9}$$

4.7.3.2 Analyse

Après l'analyse du binaire et la récupération de toutes les informations nécessaires à l'analyse, `RobustB` détermine l'ensemble des chemins de référence. Comme nous l'avons vu plus tôt, le contexte symbolique force l'exécution à suivre un seul chemin, celui correspondant à l'exécution des instructions @a, @b, @e et @f. Le PEFG correspondant au déroulage du CFG est donné dans la figure 4.17. Le chemin déterminé comme faisable par l'analyse de faisabilité de l'énumération des chemins de référence est celui apparaissant en gris sur la figure.

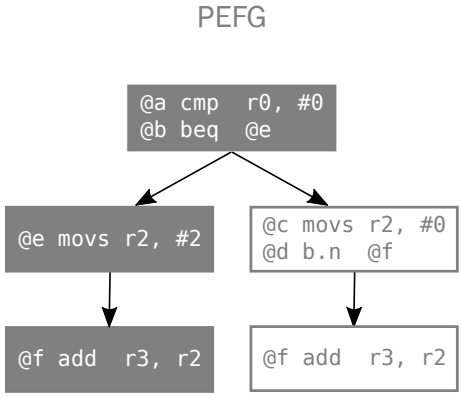

FIGURE 4.17: PEFG issu de du déroulage du CFG

C'est donc sur ce chemin que les fautes devront être injectées. Le modèle de faute étant celui du saut d'une instruction, il existe autant de points d'injection qu'il y a d'instructions sur ce chemin, *i.e.* quatre.

La première faute correspond au saut de l'instruction @a : `cmp r0, #0`. Le saut de cette instruction supprime la comparaison du registre `r0` avec la valeur 0. L'instruction

suivante (@b) utilise donc l'ancienne valeur du drapeau `Z`, celle initialisée dans le contexte symbolique. Or, sa valeur n'étant pas définie, le solveur SMT peut donc lui assigner la valeur 1 ou 0. La conséquence du fait que le drapeau `Z` soit une variable libre peut être observée sur la figure 4.18. Cette figure montre le chemin de référence, l'application de la faute, le CFG fauté issu de son application ainsi que le PEFG. Sur le PEFG, les chemins déterminés comme faisables par l'énumération des chemins fautés apparaissent en gris.

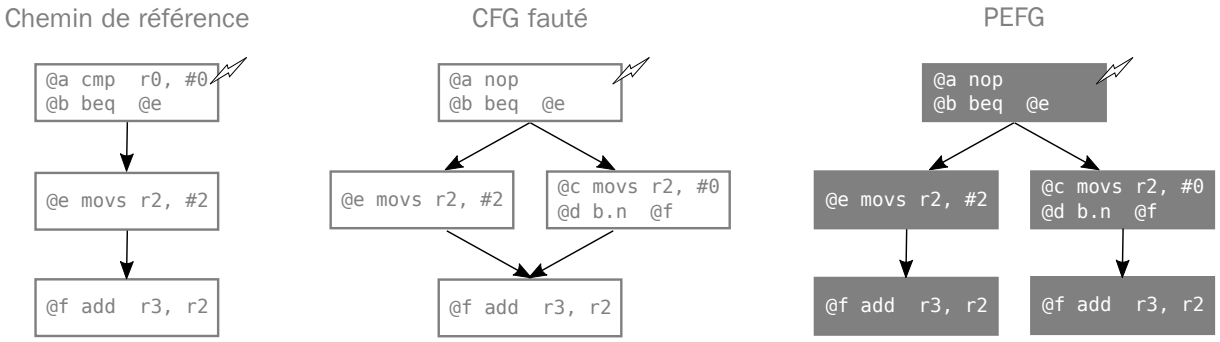

FIGURE 4.18: Chemin de référence, CFG fauté et PEFG issu de l'application du modèle de faute à l'instruction @a

Sur la figure 4.18, on voit que les deux chemins (correspondants aux séquences d'instructions {@a, @b, @e, @f} et {@a, @b, @c, @d, @f}) sont faisables. Deux problèmes de robustesse sont donc construits par `RobustB`, le premier comparant l'exécution du chemin de référence avec le premier chemin fauté et le second comparant l'exécution du chemin de référence avec le second chemin fauté.

La figure 4.19 montre une schématisation des deux problèmes de robustesse (l'annexe A détaille l'implémentation des problèmes de robustesse). On remarque d'abord que dans l'analyse de robustesse du chemin fauté correspondant aux instructions {@a, @b, @e, @f} (partie gauche de la figure), la transformation de l'instruction @b (`beq @e`), dans ce chemin où le branchement est pris, force le drapeau `Z` à la valeur 0 (valeurs finales du chemin fauté). Cette analyse de robustesse ne met pas en évidence une vulnérabilité : les deux valeurs des variables $r3_{V_f}^{\varphi r}$ et $r3_{V_f}^{\varphi f}$ sont égales. Cela signifie que, pour cette faute, si lors d'une exécution réelle la valeur du drapeau `Z` est 1 avant l'entrée dans cette portion de code, alors la faute n'a pas d'effet.

En ce qui concerne l'analyse de robustesse du chemin fauté correspondant à la séquence d'instructions {@a, @b, @c ,@d, @f} (partie droite de la figure), la transformation associée à l'instruction @b (`beq @e`) pour ce chemin force le drapeau `Z` à valoir 0 (variable $Z_{Vf}^{\varphi f}$) et les transformations associées à la séquence d'instructions du chemin fauté changent la valeur finale du registre `r3` et invalident donc la propriété.

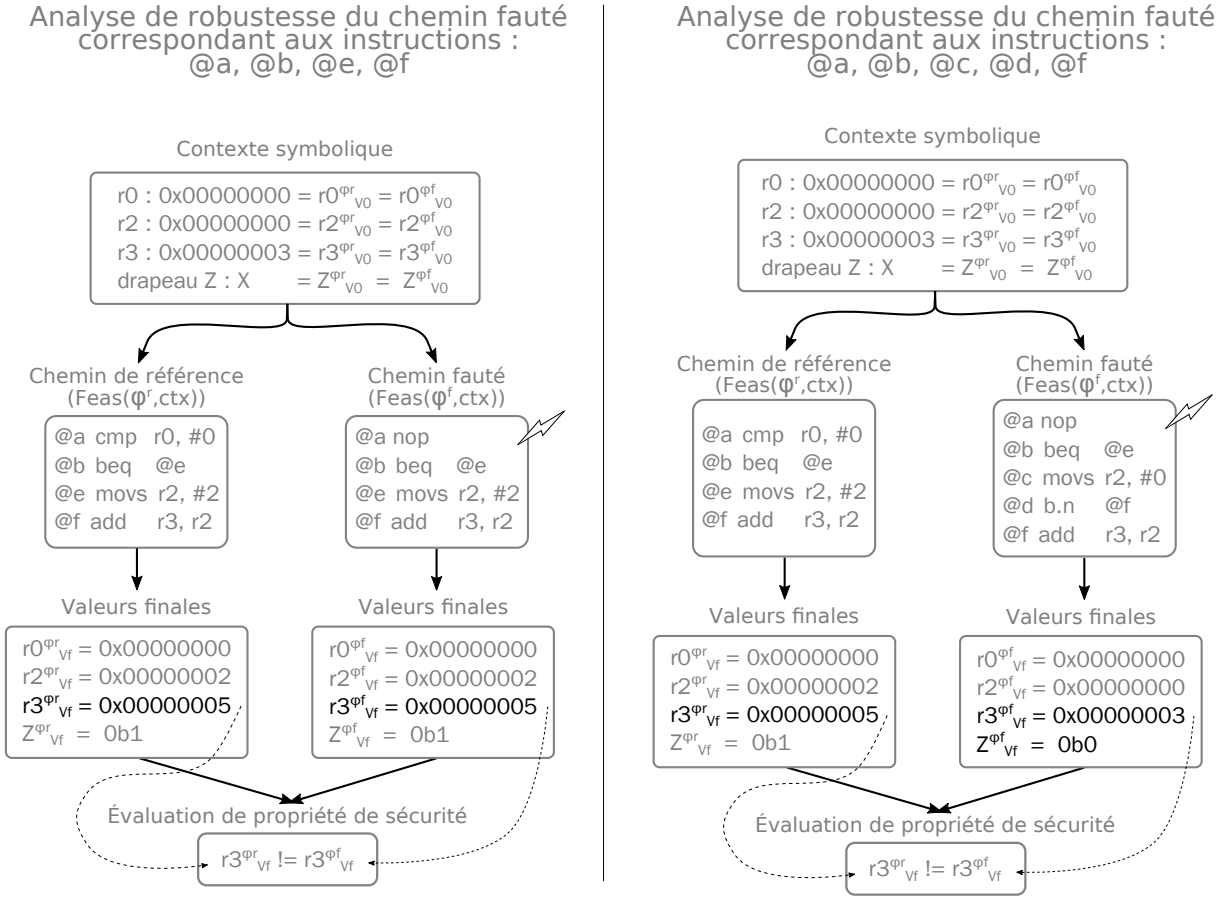

FIGURE 4.19: Analyses de robustesse des deux chemins fautés résultant du saut de l'instruction @a

Sur le chemin de référence {@a, @b, @e, @f}, trois fautes additionnelles sont possibles, à savoir : le saut des instructions @b, @e, et @f. Nous n'allons pas ici détailler ces trois fautes mais nous allons tout de même donner, pour chacune, une brève explication.

Le saut de l'instruction @b force l'exécution à suivre le chemin correspondant aux instructions {@a, @b, @c ,@d, @f} et, de la même façon que lors de la première faute, invalide la propriété. On remarque que puisque l'instruction @a a été exécutée, la valeur de la variable $Z_{V_f}^{\varphi f}$ est 0.

Le saut de l'instruction @e (`movs r2, #2`) supprime l'affectation du registre `r2` avec la valeur 2 ce qui change le résultat de l'instruction @f (`add r3, r2`) et invalide la propriété.

Le saut de l'instruction @f supprime l'assignation du registre `r3` avec le résultat de l'addition du registre `r2` et du registre `r3`, ce qui a pour effet de changer la valeur finale du registre `r3` et donc, d'invalider la propriété.

En conséquence, toutes les instructions sur le chemin de référence sont sensibles au modèle de faute du saut d'une instruction.

Le résultat de la métrique *IS* 4.6.2 est donné ci-dessous :

$$IS(@a) = P(\varphi_r) \times NV_{@a}(\varphi_r) = 1 \times 1 = 1 \tag{4.10}$$
$$IS(@b) = P(\varphi_r) \times NV_{@b}(\varphi_r) = 1 \times 1 = 1 \tag{4.11}$$
$$IS(@e) = P(\varphi_r) \times NV_{@e}(\varphi_r) = 1 \times 1 = 1 \tag{4.12}$$
$$IS(@f) = P(\varphi_r) \times NV_{@f}(\varphi_r) = 1 \times 1 = 1 \tag{4.13}$$

Toutes les instructions de l'unique chemin de référence sont sensibles, elles ont donc toutes un score IS de 1.

La valeur de la métrique *AS* 4.6.3 est donnée ci-dessous :

$$AS = P(\varphi_r) \times NV(\varphi_r) = 1 \times 4 = 4 \tag{4.14}$$

Quatre instructions sont sensibles sur l'unique chemin de référence, la surface d'attaque est donc de 4.

Le résultat de la métrique *NAS* 4.6.4 est donné ci-dessous :

$$NAS = \frac{P(\varphi_r) \times NV(\varphi_r)}{P(\varphi_r) \times NI(\varphi_r)} = \frac{1 \times 4}{1 \times 4} = 1 \tag{4.15}$$

Tous les points d'injection de l'unique chemin de référence mettent en évidence une vulnérabilité, la surface d'attaque normalisée est donc de 1.

4.8 Discussion

Dans cette section, nous comparons la méthode conçue pendant cette thèse aux travaux proches dans l'état de l'art. Nous nous intéressons aux différences qu'il peut exister entre notre méthode et celles basées sur une analyse symbolique. Trois approches remplissent ces critères [PNKI09, Jaf19, PMPD14], toutes présentées dans la section 3.3. Nous considérons des critères de comparaison qui nous semblent pertinents compte tenu de ce que nous avons présenté dans ce chapitre. Ces critères sont les suivants.

— Méthode de vérification de propriété (equivalence-checking versus propriété spécifique)

— Caractérisation du résultat : complétude et métriques

— Analyse d'une région de code vis-à-vis du contexte d'un programme englobant

Méthode de vérification de propriété (equivalence-checking versus propriété spécifique). Parmi les approches avec lesquels on se compare, celle que nous proposons ici est la seule permettant de rechercher une vulnérabilité au travers un problème d'equivalence-checking. Les autres approches permettent d'exprimer une propriété portant uniquement sur un modèle fauté (par exemple le chemin fauté dans notre approche), l'analyse par equivalence-checking permet d'exprimer la propriété en faisant référence au modèle fauté et au modèle non fauté. L'intérêt d'une analyse par equivalence-checking est particulièrement visible lorsqu'il s'agit d'une analyse symbolique qui, par nature, permet de prendre en compte tous les comportements de la région de code analysée. Dans certains cas, définir une propriété fonctionnelle vis-à-vis de valeurs attendues peut s'avérer difficile tant ses valeurs peuvent varier en fonction des entrées de l'utilisateur. Une propriété d'équivalence permet de se référer au code non fauté et ainsi elle ne nécessite pas de déterminer les relations entre les valeurs en entrée et en sortie de la région de code analysé. Toutefois, une analyse par equivalence-checking nécessite la présence du modèle fauté et du modèle non fauté, ce qui a aussi l'inconvénient d'augmenter la taille des problèmes SMT à résoudre et donc leur temps de résolution. *RobustB* permet également d'analyser un code avec une propriété spécifique (qui ne fait pas référence au modèle non fauté). On peut se poser la question de la pertinence de la présence du modèle non fauté lorsque la propriété est spécifique. Lorsque la propriété à vérifier est spécifique, la présence du chemin de référence permet de contraindre les configurations d'entrées du chemin fauté à être compatibles avec ce chemin. Cela signifie que les vulnérabilités trouvées sont aussi compatibles avec ces configurations et peuvent même y être exclusives. Pour l'illustrer, supposons que deux fautes sur des chemins de référence différents (Ra et Rb) induisent un même chemin fauté (F). Cela peut arriver lorsque Ra et Rb ont un préfixe commun et un suffixe différent dû aux configurations d'entrées. Si les deux fautes produisent des vulnérabilités qui n'apparaissent que sur l'un ou l'autre des chemins de référence, alors la présence du chemin de référence peut permettre de trouver une vulnérabilité différente pour chaque chemin. En l'absence des chemins de référence l'analyse ne trouverait qu'une seule faute correspondant à une configuration d'entrée satisfaisant soit Ra soit Rb soit les deux chemins. Une analyse réalisée en utilisant une propriété spécifique donnée peut donc permettre de trouver plus de vulnérabilités si le chemin de référence est présent que s'il est absent du problème SMT. Trouver cette vulnérabilité manquant en l'absence du chemin de référence nécessiterait de lancer une nouvelle fois l'analyse de robustesse du chemin fauté en imposant que la configuration d'entrée soit différente de celle qui a permis de trouver la première vulnérabilité. Il se peut aussi que seule une des deux fautes mène à une vulnérabilité. Aussi, la présence du chemin de référence permet d'identifier qu'une vulnérabilité n'existe que dans le cas où les configurations d'entrée satisfont l'un ou l'autre

des chemins de référence. Il y a donc des avantages à conserver le chemin de référence même lorsqu'une propriété spécifique est utilisée. Néanmoins, il serait intéressant de voir à quel point l'augmentation de la taille des problèmes SMT liée à sa présence impacte le temps de résolution des problèmes SMT. Dans tous les cas, la présence du chemin référence lors de l'utilisation d'une propriété spécifique est facultative. Dans le cas d'une analyse basée sur une propriété spécifique, le chemin de référence pourrait donc être éliminé pour améliorer les temps de résolution au détriment du nombre du vulnérabilités trouvées.

Caractérisation du résultat : complétude et métriques. La complétude d'une analyse est une information importante dans le cas où l'analyse n'a pas trouvé de vulnérabilité. En effet, l'absence de vulnérabilité seule ne permet de pas de conclure à la robustesse d'un code vis-à-vis d'un modèle de faute. La robustesse d'un code ne peut être garantie que si l'analyse est complète, c'est-à-dire, qu'elle a été en mesure d'explorer tous les comportements issus des fautes. La stratégie d'exploration des chemins fautés de `Lazart` [PMPD14] repose sur de la génération symbolique de tests. Cette dernière est réalisée avec `KLEE`, c'est donc lui qui est en charge de prendre en compte les éventuels effets des fautes sur le flot de contrôle du code fauté, notamment sur le nombre d'itérations des boucles du code. `Lazart` extrait trois réponses possibles de l'analyse faite avec `KLEE` : *vulnérable*, *robuste* et *non concluant*. La première réponse signifie qu'une vulnérabilité a été trouvée, la complétude de l'analyse n'a donc pas d'importance. Lorsque l'analyse ne trouve pas de vulnérabilité, l'analyse conclut soit à la robustesse du code (réponse *robuste* de `Lazart`) dans le cas où tous les comportements du code fauté ont été explorés, soit à l'incomplétude de l'analyse (réponse *non concluant*) dans le cas contraire. Selon les mots mêmes de l'auteur, il existe deux raisons pour que l'analyse donne un résultat non concluant suite à la génération symbolique de tests. La première est l'incomplétude de l'énumération des chemins et la seconde est l'expiration du temps imparti au solveur pour résoudre les contraintes (contraintes potentiellement trop complexes pour être résolues par le solveur SMT). L'incomplétude de l'énumération des chemins ressemble fortement à ce qui à été fait dans cette thèse. À la différence qu'une génération symbolique de tests concrétise les valeurs d'entrées avec, toutefois, une stratégie pour générer des entrées explorant tous les chemins. Dans la plupart des cas, et pour une telle stratégie d'exploration des chemins, `RobustB` ne pourrait pas conclure à la robustesse d'un code. Cela est dû au fait que les propriétés vérifiées par `RobustB` et par `Lazart` sont de natures différentes. En effet, les propriétés vérifiées par `Lazart` sont des propriétés d'accessibilité ou de non accessibilité de blocs de base tandis que celles de `RobustB` portent sur la valeur de certains éléments mémorisant au point d'observation dont l'accessibilité doit être établie. Pour `Lazart` il est donc suffisant de déterminer qu'un bloc de base est accessible tandis que les propriétés vérifiées par `RobustB` impliquent non seulement que le bloc de base où la propriété est vérifiée soit accessible mais aussi que les contraintes sur les éléments mémorisant contenues dans la propriété soient

satisfaites. Dans le cas de `RobustB`, une génération symbolique de tests, ne permettant que de trouver une unique configuration d'entrée satisfaisant l'accessibilité du bloc de base où se trouve le point d'observation, n'est pas suffisant étant donnée la nature des propriétés vérifiées par `RobustB`. En effet, la nature des propriétés vérifiées par `RobustB` implique que toutes les configurations d'entrée, satisfaisant l'accessibilité du bloc de base où se trouve le point d'observation, soient énumérées pour conclure à la robustesse de la région de code. Néanmoins, compte tenu de la différence de nature des propriétés des deux approches, la caractérisation d'incomplétude de l'énumération des chemins fautés de `Lazart` est équivalente à celle proposée par `RobustB`. À notre connaissance, l'approche proposée dans [Jaf19] ne fait pas mention de l'incomplétude de l'analyse. En effet, cette dernière utilise `LLBMC`, un outil externe, pour explorer les chemins fautés. `LLBMC` déroule les boucles un certain nombre de fois compte tenu d'une valeur par défaut, ou d'une valeur donnée à l'outil pour limiter la profondeur du déroulage. De la même façon, nous n'avons pas pu trouver d'éléments sur la complétude de l'approche proposée dans `SymPLAID` [PNKI09]. Toutefois, les auteurs indiquent dans [PNKI08], présentant l'outil sur lequel est basé `SymPLAID`, que l'approche qui y est présentée trouve toutes les "manifestations d'erreurs", ce qui suggère sa complétude.

Analyse d'une région de code vis-à-vis du contexte d'un programme englobant. Les approches opérant symboliquement sont fortement limitées par la taille du code qu'elles peuvent analyser. Elles sont donc souvent restreintes à l'analyse de petites portions de code destinées à être intégrées dans un plus gros programme. Dans `RobustB`, le contexte du programme englobant à l'entrée de la région analysée est appelé contexte symbolique. Il permet d'analyser la région de code vis-à-vis de configurations des variables d'entrée qui correspondent à des configurations possibles en entrée de la région de code lors d'une exécution réelle du programme complet. Cela peut donc permettre de conclure à la robustesse de la région de code vis-à-vis du programme englobant même si cette même région de code n'est pas robuste pour un contexte quelconque (sans aucune contrainte). Le contexte symbolique permet aussi de limiter les faux positifs. Toutefois, des faux positifs subsistent malgré la présence de ce contexte symbolique. Cela est principalement dû à ce que les aspects qui existent lors d'une exécution de code réelle, comme la pagination ou le contenu de la pile, ne sont pas modélisés dans le problème SMT. L'élimination des faux positifs serait une amélioration non négligeable de l'approche. En effet, à l'issue de l'analyse, il est pour l'instant nécessaire d'investiguer chaque vulnérabilité pour éliminer les faux positifs restants. Par exemple, une des causes de faux positifs peut être la non modélisation du placement mémoire des données et du code (plages d'adresses différentes). Dans notre approche, nous bornons les plages d'adresses accessibles correspondant aux données globales, à la pile et au code. Cela permet de limiter les faux positifs.

Afin d'éliminer les faux positifs, une analyse des résultats pourrait être réalisée : lorsque le solveur SMT trouve une vulnérabilité il fournit un contre-exemple, c'est-à-dire les valeurs des variables d'entrées de la région de code pour lesquelles la vulnérabilité à été trouvée. Une exécution réelle du programme avec ces valeurs pourrait déterminer si la vulnérabilité trouvée est un faux positif ou non. Aucune des approches avec lesquelles on se compare ici ne fait mention d'analyse en contexte de portion de code. Toutefois, un traitement des faux positifs est réalisé dans l'approche développée dans [Jaf19]. L'approche parvient à éliminer une bonne partie des faux positifs en exécutant les binaires mutants. L'exécution de ces binaires ne prend en compte qu'une configuration des entrées mais, selon l'auteur, cela permet tout de même d'éliminer la majeure partie des erreurs à l'exécution (par exemple, les erreurs de segmentation).

Enfin, dans ces travaux, nous avons choisi de construire notre propre outil dont le cœur s'apparente à un moteur d'exécution symbolique. On peut se demander pourquoi nous ne sommes pas partis d'un outil existant, notamment pour réaliser l'exploration des chemins. Nous nous sommes posé cette question au début de la thèse, notamment vis-à-vis d'`angr`, et nous avons pris la décision de réaliser cette exploration nous même pour le contrôle que cela allait nous apporter. Toutefois, une réflexion a posteriori de ce choix peut être intéressante. `RobustB` permet de faire des analyses de robustesse via de l'equivalence-checking. Cela nécessite de mettre en correspondance un chemin de référence (exécutions nominales) avec des chemins fautés (exécutions fautées découlant d'une exécution nominale) afin de pouvoir comparer le contenu de certains éléments mémorisant à la fin de ces deux chemins. Réaliser une telle analyse dans `angr` aurait nécessité de comprendre et modifier une partie conséquente du code de `angr` qui a été conçu pour adresser des problèmes différents de celui adressé par `RobustB`. Aussi, comme tout code, `angr` contient des bugs et c'est un outil qui continue d'évoluer. Il nous a semblé que développer l'approche dans `angr` nous aurait mis à la merci des potentielles évolutions (pas nécessairement en lien avec nos besoins) et surtout nous aurait rendus dépendants des choix d'implémentation réalisés. Il est à noter que nous avons dû modifier quelques parties d'`angr` (instructions mal gérées) et ces modifications se sont avérées peu simples à faire et à maintenir. Au final, il n'est pas clair que repartir de `angr` aurait été profitable. En conséquence, cela nous a donné plus d'aisance dans le développement de l'approche et peut être que certains aspects de l'approche n'auraient pas pu être approfondis si celle-ci avait été intégrée à un outil. En contrepartie, l'énumération des chemins dans `RobustB`, qui s'apparente à une exécution symbolique, est peu efficace comparée à l'état de l'art des moteurs d'exécution symbolique. En effet, l'exécution symbolique faite par `RobustB` teste la faisabilité des préfixes de chemin en repartant du point d'entrée de la région de code. Ce qui n'est pas toujours le cas pour un moteur d'exécution symbolique classique [BCD+18].

4.9 Conclusion

Dans ce chapitre, nous avons présenté la méthode d'analyse proposée pour analyser la robustesse d'un programme binaire contre des attaques en faute ainsi que son implémentation dans l'outil `RobustB`. La méthode proposée repose sur plusieurs types d'analyses (statique, symbolique et vérification formelle). La méthode formule l'analyse de robustesse par la construction et la résolution de problèmes SMT capables d'énumérer exhaustivement les configurations possibles des entrées du programme. Les problèmes des faux positifs liés à l'utilisation d'une méthode symbolique sont limités grâce à l'introduction de contextes symboliques décrivant les domaines de définitions possibles des éléments mémorisants lors d'une exécution de code réelle. Les contextes symboliques de la région de code sont issus de l'exécution symbolique du code binaire associé. La méthode autorise la représentation de modèles de faute transitoires affectant aussi bien le code que les données. Les résultats sont synthétisés sous la forme de trois métriques basées sur la pondération des vulnérabilités en fonction des probabilités d'exécutions des blocs de base le long des chemins de la région de code. Les métriques fournissent trois indicateurs sur la vulnérabilité de la région de code, elles donnent : la sensibilité de chaque instruction, la sensibilité moyenne du programme par chemin et, cette dernière ramenée au nombre d'instructions de la région de code indique la densité des vulnérabilités.

L'implémentation de la méthode dans l'outil `RobustB` vise à faciliter son usage par l'automatisation des différentes étapes du processus, notamment avec la possibilité de décrire la propriété sous forme d'annotations au niveau du code source. L'outil permet aussi d'adapter l'analyse aux spécificités d'un programme avec l'ajout de contraintes additionnelles, pouvant aussi être décrites au niveau du code source, ainsi que la symbolisation de variables durant l'exécution symbolique. `RobustB` exploite le parallélisme inhérent à l'exploration de chemins dans un programme que nous pourrons observer dans la partie 5.8 à venir.

Dans la suite, nous verrons l'application de `RobustB` sur différents types de codes, permettant, dans chaque cas, de rechercher et de mieux comprendre leurs vulnérabilités en présence de fautes.

Chapitre

5 *Validation expérimentale*

Sommaire

Ce chapitre présente différents usages de `RobustB` via sept cas d'étude et montre comment les métriques peuvent aider à comprendre les vulnérabilités, évaluer la robustesse d'un code et focaliser les efforts du développeur en charge de sécuriser une application.

Nous présentons d'abord une application de vérification de code PIN que nous utilisons dans plusieurs expériences. Viennent ensuite les expériences, organisées en six parties : les quatre premières parties correspondent à quatre usages de `RobustB` et les deux parties suivantes montrent la capacité de `RobustB` à prendre en compte des cas particuliers liés au placement du code et au code système. La première partie montre comment `RobustB` peut être utilisé pour mettre en lumière les effets des compilateurs et des niveaux d'optimisation sur les protections. La seconde partie montre comment `RobustB` peut aider à déterminer l'efficacité d'une protection lorsqu'elle est insérée à la compilation. La troisième partie montre comment l'outil et ses métriques peuvent servir à déterminer la redondance de protections face à un modèle de faute. Dans la quatrième partie, nous présentons un cas d'étude où `RobustB` détecte des chemins d'attaques. Les deux expériences suivantes sont moins des cas d'usage de `RobustB` que des illustrations de son utilisation sur des codes spécifiques : nous montrons d'abord les capacités de `RobustB` à trouver des vulnérabilités compte tenu du placement du code et des données ; puis, dans une seconde expérience, nous montrons sa capacité à analyser du code système comportant des instructions privilégiées. À la suite des expériences, nous discutons de la performance de `RobustB` ainsi que de ses limitations.

Sauf mention contraire, tous les binaires ont été produits en utilisant les compilateurs GCC 6.3 et LLVM 8.0 avec les options de compilation `-mthumb` et `-mcpu=cortex-m3`. Le solveur SMT utilisé pour toutes les expériences est Z3 [DMB08]. Les expériences ont été effectuées sur une machine contenant 40 cœurs Intel(R) Xeon(R) E5-2640 v4 à 2.40GHz avec un Hyper-Threading de 2 et ayant 126Go de mémoire. Le système d'exploitation utilisé est un Scientific Linux 7.7.

5.1 Application de vérification de code PIN

Nous avons mené des expériences sur plusieurs implémentations d'une application de vérification de code PIN nommée `VerifyPIN`. La fonction principale, de même nom, réalise l'authentification d'un utilisateur au travers du code PIN fourni par l'utilisateur. `VerifyPIN` est issu de FISSC [DPP+16] [1], un ensemble de codes écrits en C dédié à l'analyse de robustesse de codes contre les attaques en faute. FISSC fournit plusieurs versions de

1. FISSC : the Fault Injection and Simulation Secure Collection

différents programmes, une version non sécurisée de chaque programme et plusieurs autres versions combinant différentes protections contre les attaques en faute.

La fonction `VerifyPIN` implémente un service d'authentification via la comparaison de codes PIN. La version originale de la fonction appelle et vérifie la valeur de retour d'une autre fonction comparant, octet par octet, la valeur du code PIN fourni par l'utilisateur avec celle du code PIN de la carte. Les deux codes PIN sont stockés dans des tableaux d'octets et la fonction de comparaison, `byteArrayCompare`, renvoie la valeur 0 si deux octets diffèrent et la valeur 1 si les deux codes PIN sont identiques.

FISSC fournit dix versions de `VerifyPIN` numérotées de 0 à 9 ; la version 0 correspond à la version que nous appelons non protégée bien qu'elle contienne tout de même une protection, qui n'est pas explicitement citée par les auteurs, que nous appelons `HB0` dans la suite. Les 9 autres versions incluent, en plus de la protection `HB0`, différentes protections que nous présentons dans la suite. Le listing 5.1 montre le code des deux fonctions de la version $\texttt{VerifyPIN}_0$. La protection `HB0` (visible aux lignes 2, 3, 30, 33 et 36 du listing 5.1) utilise un encodage spécifique des valeurs booléennes "vrai" et "faux" : les valeurs `0x55` et `0xAA` (respectivement nommées `BOOL_FALSE` et `BOOL_TRUE`) sont utilisées en place des valeurs classiques, 0 et 1 en C, comme valeurs manipulées et retournées par la fonction `VerifyPIN`. Ces deux valeurs ont une distance de Hamming maximum sur 8 bits. L'utilisation de ces valeurs vise à complexifier le passage d'une valeur à l'autre par injection de faute : huit inversions de bits sont nécessaires pour passer de la valeur `0x55` (`0b01010101`) à la valeur `0xAA` (`0b10101010`). La variable `g_authenticated` est une variable spéciale qui peut être utilisée par un outil d'analyse pour déterminer si la fonction `VerifyPIN` a authentifié l'utilisateur. La variable `g_ptc` est un compteur d'essais indiquant le nombre d'essais possibles restants. Elle est initialisée avec un nombre, déterminant le nombre maximum d'essais de saisie du code PIN, permettant d'éviter les attaques par force brute.

Les expériences qui sont présentées dans la suite ne s'intéressent pas aux attaques visant le compteur d'essais mais uniquement à celles permettant l'authentification de l'utilisateur alors que celui-ci a saisi un mauvais code PIN. Les propriétés de sécurité utilisées dans les analyses ne font pas référence à la valeur de la variable `g_authenticated` pour déterminer l'authentification de l'utilisateur mais directement à la valeur de retour de la fonction. Les expériences ont toutes été réalisées avec une taille de code PIN (`PIN_SIZE`) égale à 4.

Les expériences présentées dans la suite ont été réalisées sur quatre versions de `VerifyPIN`. Ci-dessous, nous présentons les protections implémentées dans les différentes versions disponibles dans FISSC.

— `HB (Hardened Boolean)` : les valeurs associées aux booléens 1 (vrai) et 0 (faux)

```
1 #define PIN_SIZE 4
2 #define BOOL_TRUE 0xaa  // encodage spécifique de la valeur True
3 #define BOOL_FALSE 0x55 // encodage spécifique de la valeur False
4
5 typedef unsigned char UBYTE;
6 typedef unsigned char BOOL;
7
8 UBYTE g_userPin[PIN_SIZE];
9 UBYTE g_cardPin[PIN_SIZE];
10 BOOL  g_authenticated; // variable globale pour outils d'analyse de vulnérabilités
11 char  g_ptc; // compteur d'essais
12
13 BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size)
14 {
15         int i;
16         for (i = 0; i < size; i++) { // boucle de comparaison
17                 if(a1[i] != a2[i])
18                         return 0; // sortie anticipée
19         }
20         return 1;
21 }
22
23 BOOL verifyPIN()
24 {
25         g_authenticated = 0;
26         if (g_ptc > 0) {
27                 if (byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE) == 1) {
28                         g_ptc = 3;
29                         g_authenticated = 1;
30                         return BOOL_TRUE;
31                 } else {
32                         g_ptc--;
33                         return BOOL_FALSE;
34                 }
35         }
36         return BOOL_FALSE;
37 }
```

Listing 5.1: code de la version $\texttt{VerifyPIN}_0$

sont remplacées respectivement par `BOOL_TRUE` et `BOOL_FALSE` dans la fonction de comparaison. Cette protection est la même que la protection `HB0` sauf que cette fois, elle est appliquée à la valeur de retour de la fonction de comparaison `byteArrayCompare`.

— `FTL (Fixed Timed Loop)` : la boucle de comparaison est exécutée en temps constant. Les quatre octets des codes PIN sont tous comparés deux à deux, même après une comparaison dont le résultat indique l'inégalité de deux octets. Cela évite la sortie anticipée de boucle présente aux lignes 17 et 18 du listing 5.1. En effet, les sorties anticipées de boucle sont à proscrire car elles permettent à un attaquant de

déduire plus facilement la valeur des octets composant le code PIN de la carte.

— `INL (Inlined)` : la fonction `byteArrayCompare` est expansée dans chacun de ses sites d'appels. Plus précisément, l'appel de cette fonction dans la fonction `VerifyPIN` est remplacé par le corps de la fonction `byteArrayCompare`. Cette protection est censée rendre plus difficile le contournement de l'appel à la fonction.

— `LC (Loop Counter)` : la valeur de la variable d'induction (ou compteur d'itérations de la boucle) dans la fonction `byteArrayCompare` est testée en sortie de la boucle pour vérifier que le nombre attendu d'itérations a bien été effectué (`PIN_SIZE`).

— `DC (Double Call)` : la fonction de comparaison du code PIN (`byteArrayCompare`) de la carte avec celui entré par l'utilisateur est appelée une seconde fois lorsque le premier appel renvoie la valeur vrai. Cette protection est une forme de duplication de la comparaison des codes PIN.

— `DT (Double Test)` : la valeur de la variable contenant le résultat de la comparaison des codes PIN est testée une seconde fois lorsque le premier test (comparant sa valeur à vrai) réussit. Cette protection vise à éviter des corruptions du premier test.

— `SC (Step Counter)` : un compteur est incrémenté en des points choisis (des étapes) du programme et la cohérence de sa valeur avec celle attendue est vérifiée tout au long de l'exécution. Cette protection vise à détecter des corruptions du flot d'exécution.

— `DPTC (PIN Try Counter Decremented first)` : le compteur d'essais (`g_ptc`) est décrémenté au début de la fonction `VerifyPIN` et sa valeur est réinitialisée (à 3) lorsque la fonction a déterminé que les deux codes PIN sont identiques. Cette protection rend plus difficile le saut de la décrémentation du compteur d'essais pouvant mener à un nombre d'essais non borné.

— `PTCBK (PIN Try Counter Backup)` : le compteur d'essais est dupliqué et la valeur de chaque copie est testée. Cette protection augmente la résilience du programme à une corruption du compteur d'essais.

— `CFI (Control Flow Integrity)` : cette protection suit le schéma de Lalande *et al.* [LHB14] où des compteurs et des variables permettant de capturer des valeurs de conditions sont utilisés pour chaque structure de contrôle. L'incrémentation des compteurs et leurs vérifications, en accord avec les valeurs des conditions en lien avec le flot de contrôle, permettent de s'assurer qu'une éventuelle faute n'a pas modifié le flot de contrôle.

Le tableau 5.1 montre les protections incluses dans chaque version.

Les vérifications introduites par certaines protections ajoutent un bloc, dit de détection, qui détourne l'exécution vers du code de gestion d'erreurs lorsqu'une vérification détecte une incohérence. Bien que les codes de gestions d'erreurs ne soient pas implémentés (ce qui

TABLE 5.1: Protections incluses dans les 10 versions de `VerifyPIN`

Version	HB0	HB	FTL	INL	LC	DC	DT	SC	DPTC	PTCBK	CFI
$\texttt{VerifyPIN}_0$	•										
$\texttt{VerifyPIN}_1$	•	•									
$\texttt{VerifyPIN}_2$	•	•	•								
$\texttt{VerifyPIN}_3$	•	•	•	•							
$\texttt{VerifyPIN}_4$	•	•	•	•	•				•	•	
$\texttt{VerifyPIN}_5$	•	•	•			•			•		
$\texttt{VerifyPIN}_6$	•	•	•	•			•		•		
$\texttt{VerifyPIN}_7$	•	•	•	•			•	•	•		
$\texttt{VerifyPIN}_8$	•	•									•
$\texttt{VerifyPIN}_9$	•	•	•	•			•	•	•		•

doit être fait lorsqu'une faute est détectée est à définir pour chaque cas réel d'utilisation), si l'exécution de la fonction est détournée vers l'un de ces codes alors l'analyse doit considérer que la faute a été détectée, et qu'elle ne mène donc pas à une vulnérabilité.

Les analyses de `VerifyPIN` ont été réalisées sur 4 versions. Nous avons sélectionné 3 versions couvrant l'intégralité des protections, à l'exception de la protection CFI que nous avions jugé trop grosse pour être analysé par notre méthode, ainsi que la version 0. Les quatre versions qui sont analysées dans les expériences sont les versions 0, 4, 5 et 7.

Les analyses ont comme point d'entrée le début de la fonction `VerifyPIN` et, sauf mention contraire, le point de sortie et le point d'observation sont confondus et se situent à la fin de la fonction `VerifyPIN`. Les analyses incluent donc le code des fonctions qui sont appelées par la fonction `VerifyPIN`.

5.2 Analyse des effets de la compilation et de ses optimisations sur les protections

Dans cette partie, nous montrons comment `RobustB` peut aider à comprendre les effets du compilateur GCC et de ses niveaux d'optimisation sur des protections insérées au niveau du code source. Les 4 versions de `VerifyPIN` que nous avons sélectionnées sont considérées.

Comme déjà mentionné dans la section 5.1, les analyses recherchent les fautes pour lesquelles `VerifyPIN` renvoie la valeur associée à une authentification (`BOOL_TRUE` ou `0xAA`) alors que le code PIN saisi par l'utilisateur et celui de la carte sont différents. Une telle propriété de sécurité n'a pas besoin d'être exprimée comme une propriété d'équivalence. La

propriété de sécurité utilisée pour ces analyses est une propriété spécifique, qui ne fait référence qu'à des variables de la formule du chemin fauté. En conséquence, dans la formule d'analyse de robustesse F_{vuln}, *Vuln* exprime l'égalité de la valeur de retour (contenue dans le registre r2) avec la valeur associée à une authentification (`0xAA`) au point d'observation. La différence des valeurs des deux codes PIN (utilisateur et attendu) est exprimée par le prédicat *Fcontexte*. Nous rappelons la formule F_{vuln} vue dans la section 4.5.5 puis nous donnons l'instanciation de ses paramètres pour notre cas d'analyse.

$$F_{vuln} : \ Fcontexte \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_f \in {\varphi_f}_m^k} Feas(\varphi_f, ctx)) \wedge Vuln$$

$$F_{vuln} : \ ctx_{userpin} \neq ctx_{cardpin} \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_f \in {\varphi_f}_m^k} Feas(\varphi_f, ctx)) \wedge \ r2_{Vf}^{\varphi f} = 0xAA$$

avec $r2_{Vf}^{\varphi f}$ désignant le registre `r2` du chemin fauté φ_f de l'ensemble des valeurs finales Vf.

Les deux codes PIN ont été symbolisés lors de l'analyse du binaire. Cela permet d'abord de rechercher des vulnérabilités considérant toutes les combinaisons possibles des deux codes PIN mais aussi que les éléments du contexte symbolique ainsi que les bornes des boucles, éventuellement dépendant de la valeur des codes PIN, ne soient pas récupérées en fonction de valeurs de codes PIN spécifiques.

Les analyses ont été réalisées en considérant le modèle de faute de saut d'instruction. Nous avons réalisé deux analyses pour chacune des 4 versions de `VerifyPIN`. La première est réalisée sur un code non optimisé (compilation avec GCC et l'option `-O0`) et la deuxième sur un code optimisé (compilation avec l'option `-O2`).

TABLE 5.2: Résultats des analyses de robustesse pour quatre versions de `VerifyPIN` compilées avec `GCC` aux niveaux d'optimisation `O0` et `O2`

Version	Opt.	ANI	#RP	#inst	#vuln	AS	NAS
$\texttt{VerifyPIN}_0$	O0	73.9	4	72	96	18.37	0.25
	O2	25.3	4	29	54	10.38	0.41
$\texttt{VerifyPIN}_4$	O0	149.1	15	108	127	7.75	0.05
	O2	49	1	54	28	26	0.71
$\texttt{VerifyPIN}_5$	O0	124.2	15	128	15	1	0.01
	O2	48	1	62	8	8	0.17
$\texttt{VerifyPIN}_7$	O0	180.1	15	154	67	4.75	0.03
	O2	50	1	32	24	24	0.48

Le tableau 5.2 présente les résultats de huit analyses portant sur 4 versions `VerifyPIN`

compilées à 2 niveaux d'optimisation. La première et la deuxième colonne indiquent respectivement la version et l'option de compilation ; la colonne ANI donne le nombre moyen d'instructions par exécution ; la colonne #RP indique le nombre de chemins de référence ; la colonne #inst donne le nombre d'instructions du code binaire considéré par l'analyse ; la colonne #vuln indique le nombre de vulnérabilités ; les colonnes AS et NAS contiennent les valeurs de la surface d'attaque (AS) et de la surface d'attaque normalisée (NAS) et des deux métriques globales présentés au chapitre 4 (cf. Section 4.6).

En analysant les résultats, on peut remarquer que la version `VerifyPIN`$_5$ est la moins sensible quelle que soit la métrique considérée. Cette version est la seule à implémenter la protection DC qui introduit un nouvel appel à la fonction de comparaison lorsque le premier appel a réussi (lorsque les deux codes PIN ont été déterminés comme égaux). Doubler l'appel à la fonction de comparaison a pour but d'éliminer les vulnérabilités relevant du saut de l'appel à la fonction de comparaison. Nous pouvons donc déduire des résultats que la protection DC est celle qui offre le plus haut degré de protection face à un saut d'instruction.

L'intérêt de relier les vulnérabilités trouvées aux chemins où elles apparaissent et la probabilité d'exécution de l'instruction visée (métrique AS) peut s'illustrer en comparant les versions compilées aux niveaux d'optimisation `O0` et `O2` de la version `VerifyPIN`$_5$. En effet, les 15 vulnérabilités (`#vuln`) de la version compilée en `O0` sont dispersées sur les 15 chemins (`#RP`) donnant une surface d'attaque (AS) de 1, tandis que la version compilée en `O2` a 8 vulnérabilités et un unique chemin résultant en une surface d'attaque de 8. L'unique chemin de référence des versions 4, 5 et 7 compilées au niveau d'optimisation `O2` est la cause de l'utilisation de *blocs IT (If-Then)*[2] pour instancier le test de la fonction de comparaison. Contrairement à ce que laisse paraître le nombre de vulnérabilités trouvé, la version compilée au niveau d'optimisation `O2` est plus sensible à un attaquant qui peut injecter plusieurs fautes durant une exécution mais dont l'effet de la faute n'est pas bien maîtrisé et qui profiterait donc de plusieurs essais pendant l'exécution du code. Aussi, en comparant la surface d'attaque normalisée de ces deux analyses (colonne NAS), indiquant la densité des vulnérabilités en fonction du nombre d'instructions exécutées, on remarque que le score de la version `O0` (0,01) est plus faible que celui de la version `O2` (0,17). Ces deux valeurs peuvent être interprétées comme suit : dans la version `O0`, seul 1 % des attaques aléatoires provoquant un saut d'instruction sont réussies et dans la version `O2` ce chiffre s'élève à 17 %. Donc, un attaquant qui n'a pas la capacité de viser une instruction précise aura plus de chance de viser une instruction vulnérable dans la version compilée en `O2` que sur la version compilée en `O0`. En conséquence, quel que soit le modèle d'attaquant, la version `O0` de la version `VerifyPIN`$_5$ est la plus robuste.

2. Bloc IT : instruction permettant de conditionner l'exécution des instructions suivantes.

Si l'on compare les résultats des versions compilées au niveau d'optimisation `O0` avec leurs homologues compilés au niveau d'optimisation `O2`, on observe que la surface d'attaque normalisée a un score supérieur dans les versions `O2`. Ce phénomène est la conséquence du fait que la compilation au niveau d'optimisation `O2` produit un code plus dense (comme illustré dans la colonne `#inst`) où il y a un plus grand pourcentage d'instructions qui traitent des informations utiles. Donc, l'altération d'une instruction prise au hasard a plus de chance d'impacter le bon fonctionnement du programme.

Quant à la surface d'attaque, on observe qu'elle a un score supérieur dans les versions compilées au niveau d'optimisation `O2` sauf dans le cas de la version $\texttt{VerifyPIN}_0$. Ce phénomène s'explique d'abord par le fait que les protections des versions protégées ($\texttt{VerifyPIN}_4$, $\texttt{VerifyPIN}_5$ et $\texttt{VerifyPIN}_7$) sont altérées par la compilation au niveau d'optimisation `O2` ce qui a pour effet d'augmenter la surface d'attaque de ces versions. Ensuite, la surface d'attaque de la version $\texttt{VerifyPIN}_0$ est diminuée lorsque cette version est compilée en `O2` car, ce niveau d'optimisation tend à réduire le nombre d'instructions et diminue donc le nombre de points d'injection.

D'autre part, on observe que la surface d'attaque et la surface d'attaque normalisée des versions $\texttt{VerifyPIN}_4$ et $\texttt{VerifyPIN}_7$ compilées au niveau d'optimisation `O2` sont supérieures à celles de la versions $\texttt{VerifyPIN}_0$. Nous avons déjà évoqué ce phénomène au chapitre 2 (cf. section 2.3.2). Il s'agit du *paradoxe de la surface d'attaque* (mis en évidence par Louis Dureuil dans sa thèse [Dur16]). Ce paradoxe stipule que l'ajout de protections, censées augmenter la robustesse du code, augmente aussi le nombre d'instructions et donc le nombre de points d'injection. On observe que ce phénomène est exacerbé par la compilation au niveau d'optimisation `O2`.

Cette expérience montre l'utilité de `RobustB` et des métriques pour déterminer la version la moins sensible considérant un niveau d'optimisation et un modèle d'attaquant. On note que les métriques AS et NAS peuvent être utiles dans le cas où le compromis entre la performance du code et sa robustesse est possible. Si l'on souhaite un code robuste à 100 % alors le nombre de vulnérabilités (#vuln) suffit et il doit être de 0.

5.3 Analyse de l'efficacité d'une protection insérée à la compilation

Dans cette partie, nous montrons comment `RobustB` peut être utilisé pour déterminer la bonne insertion de protections lorsqu'elle est effectuée lors de la compilation.

5.3.1 Fonction `VerifyPIN` sécurisée à la compilation

Cette expérience a été réalisée dans le cadre du projet PROSECCO [3]. Ce projet vise l'insertion automatique de protections par le flot de compilation ainsi que la preuve de sa bonne intégration par vérifications formelles. Le compilateur utilisé dans cette expérience ainsi que `RobustB` ont tous deux été développés dans le cadre de ce projet. Le but de ces analyses est donc de s'assurer que les protections insérées par le compilateur protègent bien l'application.

Cette expérience a été conduite sur deux versions de la fonction `VerifyPIN` protégée à la compilation. Le compilateur utilisé est une version modifiée du compilateur LLVM, proposé par Barry *et al.* [BCR16], qui implémente une contre-mesure contre le saut d'instruction originellement proposé et vérifié formellement par Moro *et al.* [MDH+13]. Le schéma de protection transforme les instructions dans une forme idempotente (*i.e.* instructions qui peuvent être exécutées plusieurs fois consécutivement sans changer la sémantique du code) puis duplique chaque instruction. Appliquer cette contre-mesure à un code lui permet d'être robuste à un saut d'instruction. L'implémentation de cette contre-mesure dans le compilateur a nécessité la modification de passes de compilation existantes ainsi que l'ajout de nouvelles passes. D'abord, la passe de sélection d'instruction (visant à choisir les instructions en fonction de critères relatifs à leur performance) a été modifiée de façon à ce qu'elle privilégie les instructions idempotentes. La passe d'allocation de registres a aussi été modifiée pour spécifier qu'un même registre ne doit pas être utilisé en tant que source et destination d'une même instruction de façon à favoriser la génération d'instruction idempotente. Quatre autres passes ont été ajoutées dans le but de remplacer trois formes d'instructions du jeu d'instructions ARM Thumb-2 : les instructions d'écriture mémoire qui mettent à jour la valeur du registre utilisé pour indexer la mémoire (ce registre est donc utilisé à la fois comme source et destination), les instructions de branchement qui écrivent la valeur de retour dans un registre dédié, les instructions d'exécution conditionnelle (bloc `IT`) et enfin les instructions d'empilage et de dépilage (resp. `push` et `pop`) qui mettent en même temps à jour le pointeur de pile.

Nous avons compilé la version $\texttt{VerifyPIN}_0$ ainsi que la version $\texttt{VerifyPIN}_5$ qui, comme nous l'avons vu précédemment dans la partie 5.2, est la version la plus robuste au modèle de faute de saut d'instruction. Les deux versions de la fonction ont été compilées au niveau d'optimisation `O2` avec la version modifiée du compilateur LLVM.

Comme pour les précédentes analyses de `VerifyPIN`, nous recherchons les vulnérabilités

3. Projet PROSECCO : https ://anr.fr/Projet-ANR-15-CE39-0008. Partenaires : LIP6 Laboratoire d'Informatique de Paris 6, CEA Commissariat à l'Energie Atomique et aux Energies Alternatives

permettant une authentification alors que le code PIN saisi par l'utilisateur est incorrect. Le prédicat *Vuln* dans la formule F_{vuln} exprime l'égalité de la valeur de retour avec la valeur signifiant une authentification : $0xAA$. Les deux codes PIN ont été symbolisés et *Fcontexte* contraint le code PIN utilisateur à être différent de celui de la carte.

$$F_{vuln} \ : \ ctx_{userpin} \neq ctx_{cardpin} \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_f \in \varphi_f{}^{k}_{m}} Feas(\varphi_f, ctx)) \wedge \ r2^{\varphi f}_{Vf} = 0xAA$$

avec $r2^{\varphi f}_{Vf}$ désignant le registre `r2` du chemin fauté φ_f de l'ensemble des valeurs finales Vf.

Les analyses des deux versions face au modèle de faute de saut d'instruction n'ont révélé aucune vulnérabilité. De plus, comme le compilateur a déroulé la boucle dans les deux versions il n'y a pas de chemin frontière et l'analyse est complète. Nous pouvons donc conclure que les deux versions sont robustes à un saut d'instruction et que le compilateur a correctement déployé la protection sur les deux fonctions.

Cette expérience a permis de confirmer que la protection suffit à assurer la robustesse d'un code à un saut d'instruction et elle a aussi montré l'utilité de `RobustB` pour vérifier si les protections ont été correctement insérées à la compilation. Une telle vérification ne peut être effectuée qu'à bas niveau.

5.3.2 Fonction `memcpy` sécurisée à la compilation

Cette expérience considère une implémentation de la fonction `memcpy` où la boucle principale, réalisant la copie des données, est protégée par un schéma de protection de boucle appliqué automatiquement lors de la compilation.

Le schéma de protection considéré [PHBC17] a été établi durant la thèse de Julien Proy [Pro19]. Il vise à garantir que, en cas de faute, une boucle protégée (par le schéma de protection) est exécutée le bon nombre de fois et, sinon, que la faute est détectée. Dans cette expérience, l'unique boucle de la fonction `memcpy` a été sécurisée à la compilation par la passe de sécurisation de la boucle. La figure 5.1 montre le CFG de la fonction sécurisée. Le schéma de protection utilise deux mécanismes indépendants. Le premier stocke le registre `r2` (borne de la boucle) sur la pile avant la boucle (`@0x542`) puis recharge la valeur stockée après l'exécution de boucle (`@0x550`) pour vérifier si la valeur de la variable utilisée

pendant l'exécution de la boucle n'a pas été altérée. Le second mécanisme duplique la variable d'induction (@0x546) et l'utilise pour effectuer deux vérifications supplémentaires : une à chaque nouvelle itération de la boucle (@0x54c-@54e), qui s'assure que des itérations ne sont pas faites en trop, et une autre à la sortie de la boucle (@0x54c-@0x54e), qui teste s'il y a eu des itérations en moins. L'exécution est redirigée vers un gestionnaire de détection lorsqu'une de ces vérifications échoue.

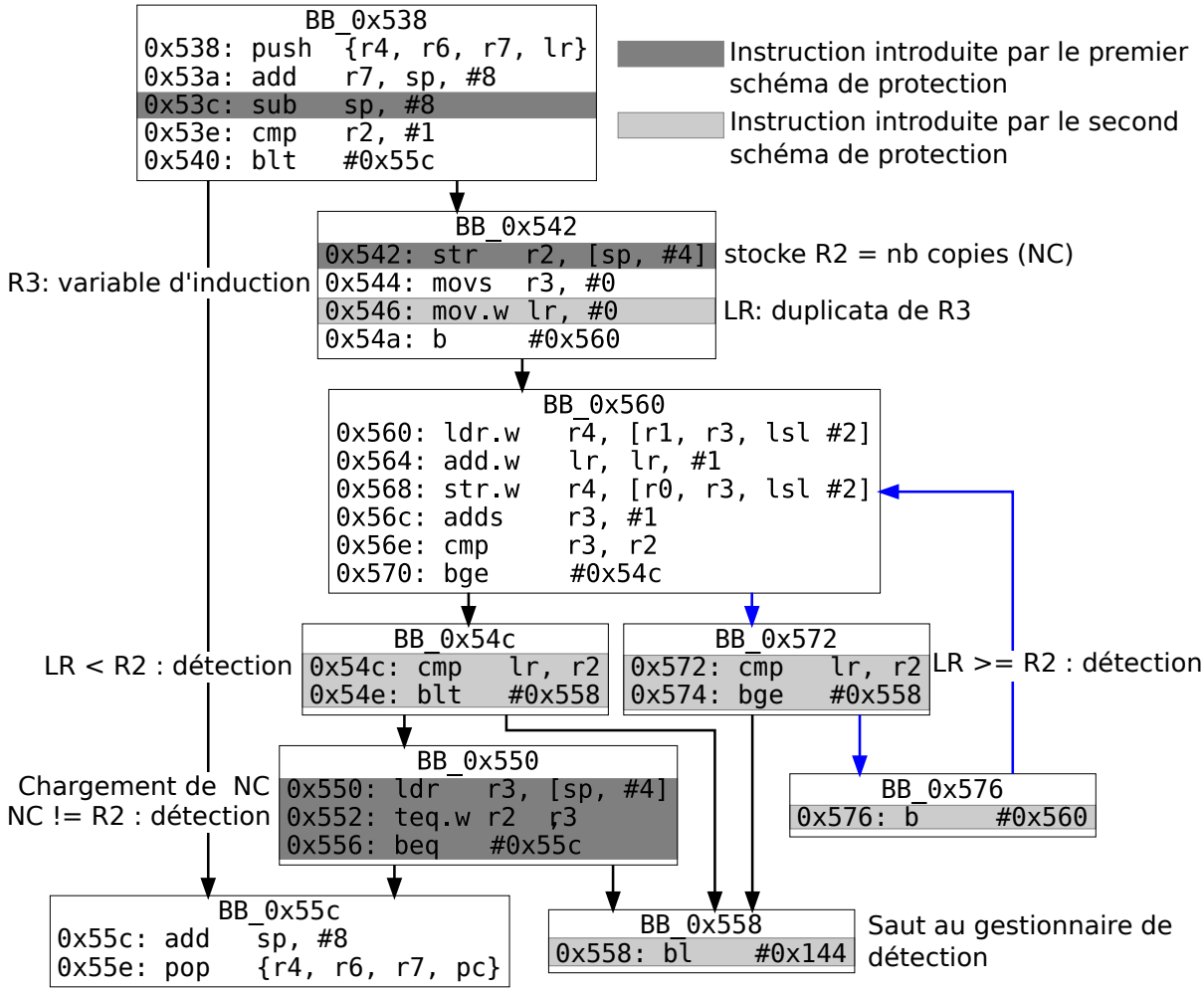

FIGURE 5.1: CFG du code assembleur de la fonction `memcpy` sécurisée à la compilation

Le schéma de protection est appliqué à la compilation, après toutes les passes d'optimisations du *middle-end* du compilateur LLVM. Comme analysé et mentionné par les auteurs, certaines optimisations effectuées en aval de l'application de la protection peuvent altérer les transformations ajoutées par l'application de la protection. Certaines passes d'optimisation (placement de code) ont été adaptées tandis que les passes d'allocation de registres et de sélection d'instructions sont restées inchangées alors que, dans certains cas particuliers, elles nécessiteraient d'être modifiées pour éviter l'altération de la protection. Cette analyse a pour but de vérifier si le code produit est robuste, *i.e.* si le back-end n'a pas affecté le schéma de protection pour ce code.

L'analyse de robustesse cherche à montrer qu'en cas de faute, la boucle effectue le bon nombre d'itérations ou que la faute est détectée. C'est pourquoi le prédicat *Vuln*, qui doit être satisfait pour que la faute soit considérée comme une vulnérabilité, exprime l'inégalité entre le nombre d'itérations exécutées par le chemin de référence et par le chemin fauté au point d'observation. Ce dernier est positionné de façon à garantir que la faute n'a pas été détectée s'il est atteint. Cette analyse a été réalisée pour la copie de deux octets et face aux modèles de faute considérés par le schéma de protection, à savoir : un saut d'instruction et une corruption de registre.

L'analyse n'a révélé aucune vulnérabilité considérant la portée de la protection annoncée par les auteurs, à savoir : un saut d'instruction ou une corruption de registre **à l'intérieur de la boucle**. Mais, l'analyse est incomplète : la corruption du registre `r2` pendant l'exécution de la boucle avec une valeur supérieure à celle qu'il contient déjà mène à de potentiels chemins inexplorés et donc à l'incomplétude de l'analyse. Toutefois, la revue du code nous permet d'assurer que ces corruptions de registre sont détectées à la sortie de la boucle. Malgré cela, un dépassement de tampons engendré par les itérations additionnelles pourrait laisser le système dans un état non désiré et être potentiellement dangereux. Un test pourrait être ajouté à l'intérieur de la boucle pour détecter plus tôt la corruption du registre `r2` et ainsi éviter un tel dépassement de tampon au prix d'une performance moindre. On peut donc considérer que le compilateur a bien appliqué le schéma de protection à ce code : les fautes engendrant un nombre anormal d'itérations sont détectées. Mais, la robustesse de cette fonction doit être établie pour chacun des codes dans lequel elle est implantée.

L'analyse a aussi révélé une vulnérabilité en dehors de la portée de la protection. Le saut de la dernière instruction de la fonction (l'instruction `pop` @`0x55e`) peut provoquer l'exécution d'une itération supplémentaire. Cette instruction est responsable de la restauration de la pile de la fonction appelante ainsi que du retour à la fonction appelante. Elle est placée à l'adresse `0x55e`, juste avant la première instruction de la boucle (@`0x560`). Le saut de l'instruction `pop` force donc l'exécution à continuer en séquence et donc, à rentrer de nouveau dans la boucle. Pendant l'exécution de l'itération supplémentaire, la variable d'induction (`r3`) et son duplicata (`lr`) sont incrémentés et sont donc maintenant supérieurs à la borne de boucle (`r2`). Le branchement conditionnel à l'adresse `0x570` détermine que la valeur contenue dans `r3` est plus grande ou égale à celle contenue dans `r2` et sort donc de la boucle. Il en est de même pour le branchement conditionnel suivant testant si la valeur de `lr` est supérieure ou égale à celle de `r2`. Enfin, l'instruction `ldr` à l'adresse `0x550` ne charge pas la valeur de la borne de boucle, préalablement stockée, mais celle du registre `r4` définie par la fonction appelante. La figure 5.2 montre l'évolution de la pile dans le cas du saut de l'instruction `pop`. La vérification située aux adresses `0x552` et `0x556` peut donc ne pas

détecter le saut d'instruction si le contenu du registre `r4`, avant l'exécution de la fonction, est égal à 1 + la valeur initiale de la borne de la boucle (correspondant au paramètre de la fonction NC dans la figure 5.1, indiquant le nombre de copies à réaliser). L'analyse de cette fonction révèle donc une vulnérabilité correspondant au saut de l'instruction `pop @0x55e`.

Aussi, comme ce code n'est pas intégré dans un plus gros programme, l'analyse a été réalisée hors contexte et le caractère exhaustif, qui a induit l'exploration des valeurs possibles du registre `r4`, a permis à `RobustB` de détecter cette vulnérabilité. Toutefois, si l'on ignore le fait que le pointeur de pile est décalé en sortie de la fonction, l'analyse en contexte d'une telle fonction, utilisée dans un programme plus gros, peut conclure sur la robustesse du programme si la valeur contenue dans la case mémoire lue est toujours inférieure au paramètre NC. Une analyse manuelle des conditions de réalisation de la vulnérabilité permet de déterminer les conditions de son exploitation. Cette analyse doit être réalisée par le développeur et il lui revient aussi de juger de sa probabilité de réalisation.

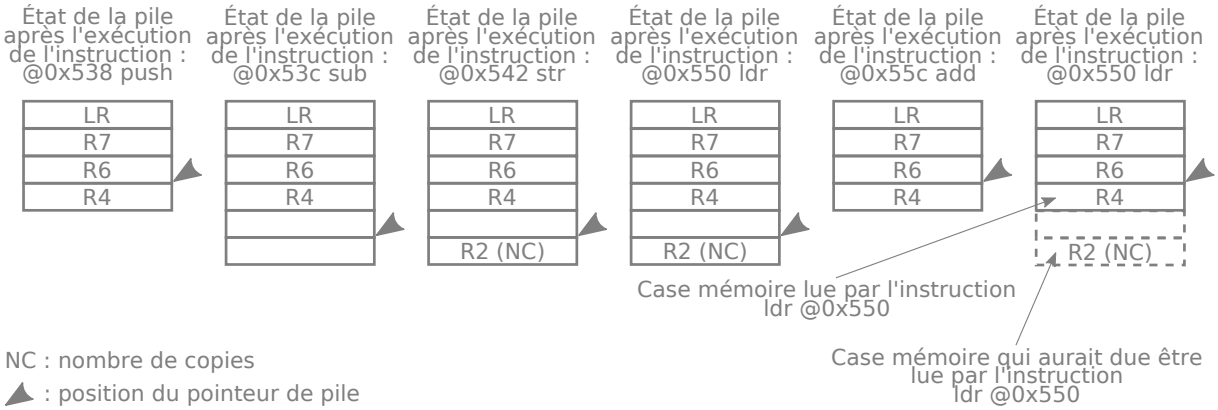

FIGURE 5.2: Évolution de la pile pour l'exécution de la fonction lors de l'injection de la faute sautant `pop @0x55e` dans la fonction `memcpy`

En conclusion, cette expérience montre l'intérêt d'une approche automatique opérant à bas niveau pour la validation de protections insérées à la compilation. Nous avons pu déterminer l'existence d'une faille exploitant le placement du code : la détection d'une telle vulnérabilité ne peut être effectuée que sur le code final.

5.4 Analyse de combinaisons de protections

Dans cette partie, nous présentons un ensemble d'expériences réalisées sur une implémentation d'une fonction de copie mémoire sécurisée par plusieurs protections indépendantes au niveau du code source. Nous montrons comment `RobustB` peut être utilisé pour déterminer la redondance des protections face à un modèle de faute et, pour comparer les effets de la compilation de deux compilateurs différents.

5.4.1 Fonction `memcpy` sécurisée au niveau source

Nous avons réalisé des expériences sur une implémentation de la fonction `memcpy` que nous avons sécurisée au niveau source. La fonction `memcpy` copie N octets depuis un tampon source vers un tampon destination. La protection et les vérifications ajoutées à la fonction visent à garantir que le bon nombre d'octets a été copié.

Le listing 5.2 montre le code de la fonction. Le schéma de protection modifie le corps de la boucle de façon à copier le contenu du tampon source dans deux tampons de destination différents : une seule copie a lieu à chaque tour de boucle, deux fois plus d'itérations sont donc nécessaires pour copier le contenu du tampon source dans les deux tampons de destination. Les copies dans les tampons de destination sont entrelacées de manière aléatoire mais les copies des éléments sont réalisées dans l'ordre du point de vue d'un tampon de destination. La boucle (ligne [25-34]) contient une structure `if-then-else` : la branche `then` réalise le transfert de la valeur d'un élément du tampon source vers le premier tampon de destination (tampon A) et la branche `else` réalise le transfert d'un élément du tampon source vers le second tampon destination (tampon B). À l'issue de l'exécution de la boucle, les branches `then` et `else` sont normalement exécutées le même nombre de fois (N fois).

L'entrelacement des copies dans les deux tampons (*i.e.*, le choix entre l'exécution de la branche `then` ou de la branche `else`) est effectué grâce à l'utilisation d'un champs de bits (`BF` dans le code). `BF` est de taille $2N$ et est contraint à contenir le même nombre de bits à 1 que de bits à 0. Plus précisément, les N bits de poids faible de `BF` sont contraints à être l'inverse des N bits suivants. À chaque itération, la valeur de son bit de poids faible détermine quelle branche (`then` ou `else`) est à exécuter. La copie du tampon source vers le tampon associé à la branche exécutée est ensuite effectuée et un compteur dédié à cette branche (nommé `iA` pour le tampon A, et `iB` pour le tampon B) est incrémenté. À l'issue de l'exécution de l'une ou l'autre des branches, `BF` est décalé d'un bit vers la droite.

Après l'exécution de la boucle, quatre vérifications ont lieu : elles ont pour but de détecter si le nombre de copies dans chaque tampon est correct. Ces quatre vérifications (nommée C1, C2, C3 et C4) testent des propriétés reflétant le bon déroulement de l'exécution :

- C1 : vérifie que `BF` vaut 0 (conséquence des décalages successifs vers la droite)
- C2 : vérifie que la variable d'induction est égale à $2N$
- C3 : vérifie que les valeurs des compteurs `iA` et `iB`, respectivement associés au nombre de copies vers le tampon A et vers le tampon B, sont égales
- C4 : vérifie que les compteurs `iA` et `iB` contiennent la valeur N

```
1  #include <stdint.h>
2  #include <stdlib.h>
3
4  #define NOK 0xdeadc0de
5  #define OK  0xab15b055
6
7  uint32_t secured_memcpy(void * dA, void * dB, const void * src, size_t N)
8  {
9          uint32_t i, iMax,iA, iB;
10         uint32_t BF, Pat, Mask, Shift;
11         uint32_t randSeed = 1;
12
13         iMax = N; // = 3
14         iA =  iB = 0;
15
16         Mask = 0xFFFF >> (16 - N); // = 0x7
17         Shift = 16 - (16 - N); // = 3
18
19         Pat = randSeed & Mask; // = 0x1 & 0x7 = 0x1
20         BF = ((~Pat & Mask) << Shift) | Pat; // = 0x30 | 0x1 = 0b110 001
21
22         // Boucle principale
23         for (i = 0; i < (iMax << 1); i++) {
24                 if ((BF & 1) == 0) { // Copie dans le tampon A
25                         ((char *)dA)[iA] = ((char *)src)[iA];
26                         iA += 1;
27                 } else { // ((BF & 1) == 1) : Copie dans le tampon B
28                         ((char *)dB)[iB] = ((char *)src)[iB];
29                         iB += 1;
30                 }
31                 BF >>= 1;
32         }
33
34         if (BF != 0)
35                 return NOK;
36
37         if (i != (iMax << 1))
38                 return NOK;
39
40         if (iA != iB)
41                 return NOK;
42
43         if ((iA != iMax) || (iB != iMax))
44                 return NOK;
45
46         return OK;
47 }
```

Listing 5.2: code la fonction `memcpy`

La fonction renvoie `NOK` si l'un de ces tests échoue et `OK` sinon.

Dans ces analyses, nous recherchons les vulnérabilités pour lesquelles la fonction renvoie la valeur `OK` alors que la faute a induit un mauvais nombre d'itérations de la boucle principale. Pour la propriété, nous avons besoin de déterminer le nombre d'itérations indépendamment des variables d'induction. Pour cela, nous avons attaché un compteur à la première instruction de la boucle dans les deux chemins présents dans la modélisation SMT, noté *nbiter*. La propriété de sécurité utilisée est donc une propriété d'équivalence portant sur les compteurs *nbiter*. Dans la formule F_{vuln}, *Vuln* est un prédicat exprimant l'inégalité du nombre d'itérations effectuées dans le chemin de référence ($nbiter_{Vf}^{\varphi r}$) et dans le chemin fauté ($nbiter_{Vf}^{\varphi f}$).

$$F_{vuln} \ : \ Fcontexte \wedge Feas(\varphi_r, ctx) \wedge (\bigvee_{\varphi_f \in \varphi_f{}_m^k} Feas(\varphi_f, ctx)) \wedge \ nbiter_{Vf}^{\varphi r} \neq nbiter_{Vf}^{\varphi f}$$

avec $Fcontexte = true$ et $nbiter_{Vf}^{C}$ désignant le nombre d'itérations réalisées dans le chemin C de l'ensemble des valeurs finales Vf.

Les analyses ont été réalisées pour tous les entrelacements possibles des copies dans les tampons, pour les modèles de faute de saut d'instruction et de corruption de registre et considérant deux compilateurs : GCC et LLVM.

5.4.2 Analyse face au modèle de faute de saut d'instruction

Dans cette expérience, nous avons analysé les binaires issus de la compilation de la fonction `memcpy` au niveau d'optimisation `O2` avec les compilateurs GCC et LLVM.

Les deux analyses n'ont révélé aucune vulnérabilité. Une analyse manuelle des codes binaires désassemblés a montré que la vérification C2 et la moitié de la vérification C4 ont été supprimées par les compilateurs. Nous pouvons donc conclure que la vérification C2 et la partie supprimée de la vérification C4 ne sont pas nécessaires pour garantir le bon nombre d'itérations de la boucle de la fonction face au modèle de faute de saut d'instruction.

Pour identifier les potentielles redondances des vérifications face à un saut d'instruction, nous avons réécrit quatre nouvelles versions de la fonction où chacune des versions contient une seule vérification. On note VCi avec $i \in [1, 4]$ la version dans laquelle seule la vérification Ci est implémentée et VCa la version dans laquelle toutes les vérifications sont présentes. Pour ces analyses, nous avons compilé les cinq versions au niveau d'optimisation `O2` avec

les deux compilateurs. La revue de code des binaires montre que comme pour la version VCa, déjà analysée, la vérification C2 et la moitié de la vérification C4 sont absentes dans les binaires des versions VC2 et VC4 respectivement.

TABLE 5.3: Analyses de robustesse des dix versions de la fonction `memcpy` sécurisée, compilées avec GCC et LLVM au niveau d'optimisation `O2` assembleur issus face au modèle de faute de saut d'instruction

Version	**compilateur**	**ANI**	**#RP**	**#inst.**	**#vuln**	**AS**	**NAS**
VCa	GCC	77	8	44	0	0	0
VC1	GCC	76	8	38	52	5.5	0.09
VC2	GCC	72	8	34	81	10.13	0.14
VC3	GCC	74	8	38	6	0.75	0.01
VC4	GCC	78	8	44	0	0	0
VCa	LLVM	77	8	125	0	0	0
VC1	LLVM	74	8	114	16	2	0.03
VC2	LLVM	68.5	8	108	45	5.6	0.08
VC3	LLVM	72.5	8	116	14	1.75	0.02
VC4	LLVM	74.5	8	118	0	0	0

Le tableau 5.3 présente les résultats de dix analyses de robustesse de la fonction `memcpy` face au modèle de faute de saut d'instruction. La première et la deuxième colonne indiquent respectivement la version et l'option de compilation ; la colonne ANI donne le nombre moyen d'instructions par exécution ; la colonne #RP indique le nombre de chemins de référence ; la colonne #inst donne le nombre d'instructions du code binaire considéré par l'analyse ; la colonne #vuln indique le nombre de vulnérabilités ; les colonnes AS et NAS contiennent les valeurs de la surface d'attaque (AS) et de la surface d'attaque normalisée (NAS), les deux métriques globales présentés au chapitre 4 (cf. section 4.6).

Les résultats montrent que les analyses des versions VCa et VC4 n'ont révélé aucune vulnérabilité face au modèle de faute de saut d'instruction (colonne #vuln). Puisque la version VC2 ne contient aucune vérification après compilation, celle-ci apparaît comme étant la plus vulnérable quelle que soit la métrique.

En ce qui concerne l'analyse de complétude de l'énumération des chemins fautés, les analyses de robustesse des versions VCa GCC, VCa LLVM, VC4 GCC et VC4 LLVM sont incomplètes. Le nombre de fautes menant à de potentiels chemins inexplorés est compris entre 23 et 31 inclus. Nous avons réalisé des analyses additionnelles sur ces quatre versions en relâchant les bornes des boucles, de façon à autoriser une itération supplémentaire lors de l'énumération des chemins fautés (cf. section 4.4.3). Ces analyses n'ont pas révélé de vulnérabilités supplémentaires et l'analyse de complétude de l'énumération des chemins fautés n'a révélé que deux potentiels chemins de moins dans la nouvelle version VC4

compilée avec GCC. La vérification C4, testant la valeur des compteurs avec la borne de boucle, devrait détecter d'éventuelles itérations supplémentaires une fois sortie de la boucle, mais les écritures au-delà du tampon de destination effectuées lors de l'exécution des itérations supplémentaires pourraient laisser le système dans un état non prévu. Une autre version du code doit être créée s'il est déterminé qu'un tel état peut nuire à la sécurité du système.

Les métriques AS (surface d'attaque) et NAS (surface d'attaque normalisée) montrent que les binaires produits par le compilateur LLVM sont, en moyenne, plus robustes que ceux produits par le compilateur GCC. Une comparaison manuelle des binaires produits par ces deux compilateurs a révélé que dans toutes les versions produites par LLVM, la boucle est déroulée quatre fois et est immédiatement suivie par un épilogue réalisant une, deux ou trois itérations. Le déroulage de la boucle est visible dans la colonne `#inst` indiquant le nombre d'instructions du programme. Cette optimisation du compilateur semble réduire le nombre de points d'injection induisant une vulnérabilité.

Cet effet peut être illustré si l'on considère la version VC1. La métrique IS [4] indique que deux instructions sont responsables de $12/13^{\text{ième}}$ de la surface d'attaque dans la version VC1 compilée avec GCC. Ces deux instructions incrémentent la variable d'induction de la boucle dans les deux branches. Le saut d'une de ces deux instructions durant une itération de la boucle principale induit automatiquement l'exécution d'une itération supplémentaire. Pour la version VC1 compilée avec LLVM, la métrique IS indique que deux instructions sont responsables des vulnérabilités impliquant l'exécution d'une itération supplémentaire. La première instruction incrémente la variable d'induction à la fin de la boucle déroulée. Puisque la boucle est déroulée dans la version compilée avec LLVM, l'instruction incrémentant la variable d'induction est exécutée moins de fois que dans la version compilée avec GCC. La seconde instruction est responsable du calcul du nombre d'itérations de la boucle déroulée. Cette opération n'a lieu qu'une seule fois, juste avant l'exécution de la boucle. Nous attribuons aussi au déroulage de boucle la différence observée entre les scores des métriques (AS et NAS) des deux versions incluant la vérification VC2.

Quant aux deux versions VC3, les métriques (AS et NAS) indiquent que celle compilée avec LLVM est plus vulnérable que son homologue compilée avec GCC. Dans la version compilée avec LLVM, la métrique IS indique qu'une instruction est sensible sur tous les chemins. La présence de cette vulnérabilité semble être due à une mauvaise optimisation du compilateur LLVM. L'instruction mise en cause se situe au début du programme, elle compare le paramètre N avec la valeur 0 dans le but de ne pas réaliser de copies si N vaut 0. Il se trouve que le registre `r3`, contenant ce paramètre, est le même que le registre

4. La métrique IS n'est pas représentée dans les tableaux étant donné que cette métrique est à la granularité d'une instruction

qui est utilisé en tant que compteur du nombre de copies dans un des deux tampons. Ce registre est testé avec la valeur de l'autre compteur à la fin de l'exécution de la fonction (vérification C3). Dans la branche qui est exécutée lorsqu'il est déterminé que le paramètre N (contenu dans le registre `r3`) vaut 0, le compteur du nombre de copies, contenu aussi dans le `r3`, est mis à 0. Or, c'est cette mise à zéro qui ne permet pas à la vérification C3 de détecter la faute. Il n'y a, a priori, aucune raison d'assigner 0 à un registre dans une branche qui est exécutée seulement si ce registre vaut 0. Ce problème ne se pose pas dans la version compilée avec GCC, le placement de code, le contenu des blocs et les instructions utilisées n'ont rien de commun avec ce qui est présent dans la version compilée avec LLVM. Il existe plusieurs façons, nécessitant peu d'efforts, qui permettent de supprimer cette vulnérabilité, par exemple : supprimer l'instruction qui met à 0 le registre `r3`, modifier une instruction en amont de l'instruction fautée de façon à positionner les drapeaux pour que l'exécution prenne l'autre branche si la comparaison du paramètre N avec la valeur 0 est fautée. C'est donc une combinaison de décisions au niveau du placement du code et de manquement dans l'optimisation de l'allocation de registres du compilateur LLVM qui a permis l'existence de cette vulnérabilité.

En résumé, l'analyse de robustesse a révélé que la version intégrant toutes les vérifications est aussi robuste qu'une version n'intégrant qu'une partie des vérifications (VC4) face à une faute de type saut d'instruction. La métrique IS, remontant les vulnérabilités au niveau des instructions, nous a aidé à comprendre la source des vulnérabilités des différentes versions.

5.4.3 Analyse face au modèle de faute de corruption de registre

Nous avons réalisé une seconde analyse sur les mêmes versions de la fonction face au modèle de faute de corruption de registre. L'analyse de la version VCa compilée avec GCC a révélé 26 vulnérabilités. 24 d'entre elles concernent la corruption du registre contenant la valeur du paramètre N (le nombre d'octets à copier) avant sa première utilisation. Les corruptions des paramètres avant leur première utilisation sont équivalentes à la corruption des paramètres avant l'appel de la fonction et doivent donc être gérées au niveau de la fonction appelante (par exemple en dupliquant le paramètre). Dans la suite, nous évaluons la robustesse de la fonction `memcpy` sans prendre en compte les fautes correspondant à la corruption des paramètres avant leur première utilisation. En conséquence, la colonne `#vuln` du tableau 5.4, présentant le résultat des analyses, comptabilise uniquement les vulnérabilités qui ne sont pas de ce type.

Le tableau 5.4 présente les résultats de dix analyses de robustesse de la fonction `memcpy` face au modèle de faute de corruption de registre. La première et la deuxième colonne

TABLE 5.4: Résultats des analyses de robustesse des dix versions de la fonction `memcpy` sécurisée au niveau source, compilées avec GCC et LLVM au niveau d'optimisation `O2` et face au modèle de faute de corruption de registre.

Version	**compilateur**	**ANI**	**#RP**	**#inst**	**#vuln**	**AS**	**NAS**
VCa	GCC	1540	8	44	2	1.25	0.001
VC1	GCC	1520	8	38	177	117.5	0.081
VC2	GCC	1440	8	34	224	137	0.1
VC3	GCC	1480	8	38	102	57.5	0.04
VC4	GCC	1560	8	44	20	6.5	0.004
VCa	LLVM	1540	8	125	0	0	0
VC1	LLVM	1480	8	114	64	47	0.033
VC2	LLVM	1370	8	108	88	53.0	0.041
VC3	LLVM	1450	8	116	42	12.7	0.009
VC4	LLVM	1490	8	118	14	7.25	0.005

indiquent respectivement la version et l'option de compilation ; la colonne ANI donne le nombre moyen d'instructions par exécution ; la colonne #RP indique le nombre de chemins de référence ; la colonne #inst donne le nombre d'instructions du code binaire considéré par l'analyse ; la colonne #vuln indique le nombre de vulnérabilités ; les colonnes AS et NAS contiennent les valeurs de la surface d'attaque (AS) et de la surface d'attaque normalisée (NAS), les deux métriques globales présentées au chapitre 4 (cf. section 4.6).

L'analyse de la version VCa compilée avec GCC n'a révélé que deux vulnérabilités. Ces deux vulnérabilités correspondent à une corruption de la valeur de N impactant à son tour le champ de bits (BF) et résultant en la non-exécution d'une itération. La revue du code binaire de la version VCa compilée avec LLVM nous a permis de comprendre que le compilateur a effectué un ordonnancement différent des instructions ayant pour effet de supprimer ces deux vulnérabilités. Cette version est la seule robuste a une corruption de registre. Cependant la version VCa compilée avec GCC pourrait aussi être robuste en ne changeant que très peu l'ordre des instructions. Toutes les autres versions contiennent un nombre significatif de vulnérabilités. Comme pour l'analyse face au modèle de faute de saut d'instruction, les versions compilées avec le compilateur LLVM ont leur boucle déroulée, et tendent donc à être plus robustes que leurs homologues compilées avec GCC. Ici, aucune vérification seule ne permet d'atteindre le niveau de robustesse de la version VCa compilée avec LLVM.

Comme une seule vérification n'est pas suffisante, nous avons analysé des nouvelles versions de la fonction intégrant la combinaison de deux vérifications parmi C1, C3 et C4 [5] compilées avec le compilateur LLVM au niveau d'optimisation `O2`. L'analyse de la version

5. La vérification C2 étant tout le temps éliminée par les compilateurs, nous ne l'avons pas considérée

combinant les vérifications C1 et C4 n'a révélé aucune vulnérabilité face au modèle de faute de corruption de registre. Nous pouvons donc conclure que cette version est équivalente à la version VCa compilée avec LLVM face au modèle de faute de corruption de registre et que la vérification C3 est inutile.

Les analyses de complétude de l'énumération des chemins fautés de la version VCa compilée avec LLVM et pour celle combinant les vérifications C1 et C4 compilée avec LLVM indiquent que 80 fautes mènent à de potentiels chemins inexplorés. Nous avons analysé la robustesse des deux versions en autorisant une itération supplémentaire et ces deux analyses ont le même résultat : aucune nouvelle vulnérabilité n'a été trouvée et le même nombre de potentiels chemins inexplorés reste inchangé (80). Comme lors de l'expérience réalisée face au modèle de faute de saut d'instruction, les vérifications devraient détecter les itérations supplémentaires à la fin de la boucle, mais les dépassements de tampon induits par ces itérations pourraient mettre à mal la sécurité du système. `RobustB` ne peut donc pas conclure sur la robustesse de ces versions. Une solution externe doit être employée ou une nouvelle version de la fonction doit être pensée avec pour objectif d'éviter la génération de chemins frontières (comme l'ajout d'une vérification à l'intérieur de la boucle) pour être ensuite analysée par `RobustB`.

En résumé, ces expériences montrent l'impact de deux compilateurs (suppression et altération de vérifications) et plus particulièrement l'impact des passes de sélection et de placement des instructions sur la robustesse du code binaire. Cela montre la nécessité de réaliser une analyse de robustesse sur le code assembleur ou binaire. Nous avons pu déterminer que seules deux vérifications sur les quatre suffisent à l'obtention du même niveau de robustesse face aux modèles de faute de saut d'instruction et de corruption de registre. En ce qui concerne la corruption du paramètre N avant sa première utilisation, toutes les versions (même la version VCa compilée avec LLVM) sont sensibles et nécessitent donc l'intégration de protections dans la fonction appelante pour être totalement robustes à une corruption de registre.

5.5 Détection de chemins d'attaques

Dans le cadre du projet PROSECCO, nous avons réalisé une expérience ayant pour but la découverte de chemins d'attaque.

Cette expérience a été réalisée en collaboration avec une équipe du CEA LIST basée à Gardanne. Pour cette partie du projet, le CEA LIST était en charge de valider, au travers d'expérimentations réelles, la bonne intégration des protections par le compilateur

PROSECCO. Les injections de fautes réelles ont été réalisées via l'utilisation d'un laser, sur une plateforme `STM32F1000RBT6` [6] contenant un microcontrôleur `STM32bit` composé d'un processeur ARM Cortex-M3, d'une mémoire SRAM et d'une mémoire Flash. La caractérisation [7] de cette plateforme a mis en évidence sa sensibilité à une attaque visant la mémoire Flash lors de la lecture d'une instruction. Cette attaque permet de mettre à 1 un bit dans l'encodage d'une instruction lors de sa lecture. Cette faute est équivalente au modèle de faute d'instruction bit-set transitoire (présentée au chapitre 4 cf. sections 4.4.2 et 4.5.3). En effet, c'est la mémoire flash qui contient le code lors de son exécution (il n'y a pas de cache). Pour exploiter les fautes réalisables par injections laser, le CEA LIST a besoin de connaître les fautes de type instruction bit-set qui mènent à une vulnérabilité. Nous avons donc proposé d'utiliser `RobustB` pour effectuer cette recherche.

L'analyse a été réalisée sur la version $\texttt{VerifyPIN}_5$ protégée à la compilation par le compilateur PROSECCO (présentée dans la partie 5.3.1). Cette version n'est pas, a priori, robuste à une faute de type instruction bit-set. Comme pour l'expérience réalisée dans la partie 5.3.1, cette analyse recherche les vulnérabilités permettant une authentification alors que le code PIN saisi par l'utilisateur est incorrect et la formule F_{vuln} est identique aux autres analyses réalisées sur le code des différentes versions de `VerifyPIN` :

$$F_{vuln} \ : \ ctx_{userpin} \neq ctx_{cardpin} \wedge Feas(\varphi_r, ctx) \wedge \Big(\bigvee_{\varphi_f \in \varphi_f{}^k_m} Feas(\varphi_f, ctx) \Big)$$
$$\wedge \quad r2^{\varphi f}_{Vf} = 0xAA$$

avec $r2^{\varphi f}_{Vf}$ désignant la registre `r2` du chemin fauté φf de l'ensemble des valeurs finales Vf.

L'analyse a révélé de multiples vulnérabilités. Nous en avons sélectionné une : elle correspond à une instruction qui n'est exécutée qu'une seule fois tout au long de l'exécution de la fonction et dont la vulnérabilité associée ne nécessite pas de conditions particulières sur les entrées (c'est-à-dire qu'elle ne nécessite pas de valeurs spécifiques des codes PIN). L'instruction en question est la suivante : `movw r1, #x29b0`. Elle charge, dans le registre `r1`, les 16 bits de poids faible de l'adresse du code PIN attendu. `RobustB` a détecté que, le modèle de faute instruction bit-set appliqué au 19ème bit de cette instruction, met en évidence une vulnérabilité. La mise à 1 du 19ème bit de cette instruction ajoute 4 à

6. Fiche technique : https ://www.st.com/resource/en/datasheet/stm32f100cb.pdf (17/01/2020)
7. Caractérisation : exploration des paramètres du moyen d'injection (spatiale, temporelle, puissance, ...) pour mettre en évidence des modèles de faute.

l'adresse qui est stockée dans le registre `r1`. Les encodages binaires et leur désassemblage avant et après la mise à 1 du 19ème bit de l'instruction en question sont les suivants :

— 0001000110110→0←001111011001000010 : `movw r1, #0x29b0`

— 0001000110110→1←001111011001000010 : `movw r1, #0x29b4`

Or, cette nouvelle adresse, contenue dans le registre `r1`, est la même que celle contenue dans le registre `r3`, contenant l'adresse du code PIN utilisateur. Cette faute impacte donc l'exécution du code car toutes les comparaisons des deux codes PIN sont faites entre le code PIN utilisateur et lui-même.

Le CEA LIST avait besoin d'un certain nombre d'informations pour pouvoir réaliser l'attaque, nous lui avons donc fourni les vulnérabilités sous le format présenté dans la figure 5.3.

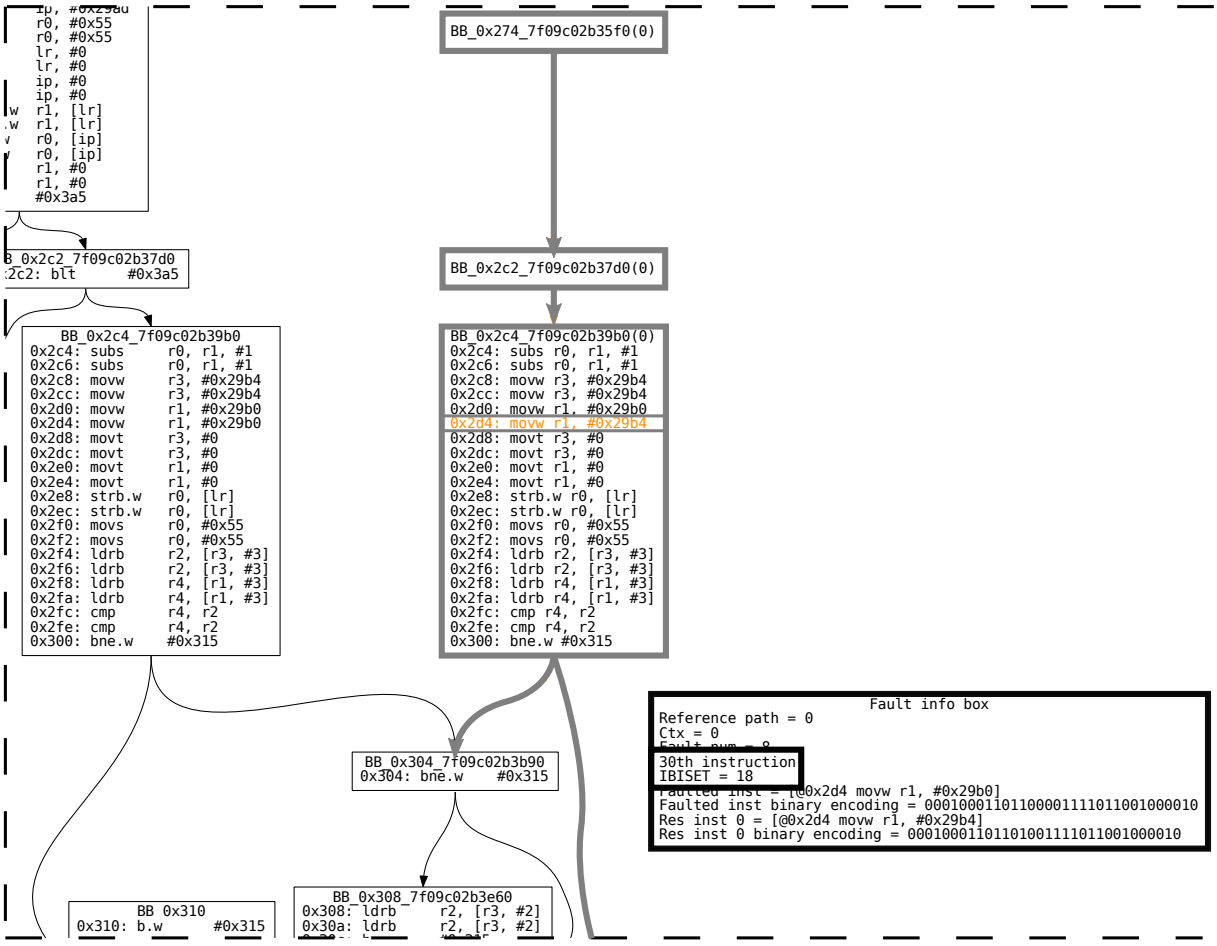

FIGURE 5.3: Vulnérabilité de la fonction $\texttt{VerifyPIN}_5$ sécurisée face au modèle de faute d'instruction bit-set

La figure 5.3 montre une partie du CFG modifié de la version $\texttt{VerifyPIN}_5$ ainsi que les informations nécessaires à la reproduction de la faute par expérimentations réelles. Le CFG est modifié par l'ajout du chemin suivi jusqu'à l'injection de faute (correspondant aux blocs aux contours grisés sur la figure 5.3). L'instruction fautée après application de la

faute est encadrée en gris sur un bloc du chemin (`movw r1, #0x29b4`). Les informations nécessaires à l'application de la faute sont disponibles dans la boîte intitulée *Fault info box*. Les informations nécessaires à l'expérimentation réelle, présentes dans la *Fault info box*, sont : le numéro de l'instruction (*30th instruction* sur la figure 5.3) et le numéro du bit de l'instruction à fauter (*IBITSET = 18* sur la figure 5.3).

Utilisant les mêmes paramètres que lors de la caractérisation du modèle de faute d'instruction bit-set transitoire et en visant le 19ème bit de la 30ème instruction, le CEA LIST a pu reproduire cette faute.

En conclusion, cette expérience a montré que `RobustB` peut être aussi utilisé du point de vue de l'attaquant pour la recherche de vulnérabilités qui peuvent ensuite être exploitées via des injections de fautes réelles.

5.6 Présence de données dans le code

Comme expliqué au chapitre 4 (cf. section 4.7.1.5), en ARM Thumb-2, le compilateur peut placer des données entre deux instructions d'une fonction. Ce comportement a pour but d'améliorer la performance et de réduire la taille du code mais il peut induire des vulnérabilités si les données sont interprétées comme des instructions et exécutées, par exemple, à la suite d'un saut d'instruction. Cette expérience est réalisée sur un code ayant cette caractéristique.

Cette expérience a été conduite sur une version modifiée de la version ($\texttt{VerifyPIN}_0$). Le code source C de la fonction a été modifié de telle sorte que le compilateur soit forcé de placer des données au milieu du code de la fonction.

Les fonctions de petites tailles ont leurs données placées à la suite de leurs instructions. Toutefois, la taille d'encodage du décalage (*offset*) relatif peut être trop petite pour encoder la distance séparant l'instruction de lecture des données lorsque celle-ci est trop importante. La fonction $\texttt{VerifyPIN}_0$ n'est pas assez longue pour que les données soient placées au milieu du code, nous avons donc augmenté artificiellement la taille du code pour forcer le compilateur à placer les données au milieu des instructions de la fonction.

Pour cette expérience, la fonction $\texttt{VerifyPIN}_0$ a été compilée avec une taille de code PIN de 2. Le listing 5.3 montre deux parties des instructions assembleur du code de la fonction $\texttt{VerifyPIN}_0$ modifiée incluant la donnée sous une forme désassemblée (la donnée est affichée sous la forme de deux instructions de 16 bits aux lignes 3 et 4). Le compilateur

a placé la donnée à la suite de la boucle comparant les codes PIN et avant les instructions réalisant la comparaison du nombre d'octets identiques (valeur contenue dans le registre `r4`) avec la valeur attendue (2). Le branchement conditionnel suivant cette comparaison détermine la valeur de retour de la fonction : la valeur "1" signifie que les codes PIN sont identiques et la valeur "0" signifie qu'ils sont différents. Comme on peut le voir dans le listing 5.3, si la donnée désassemblée est exécutée, elle est alors interprétée comme une séquence de deux instructions : une instruction `strh` encodée sur 16 bits (permettant le stockage d'un demi-mot dans la mémoire) suivie d'une instruction `movs` encodée sur 16 bits (initialisant un registre avec un immédiat). L'instruction `movs` écrit la valeur `2` dans le registre `r4`, or le registre `r4` est utilisé par la fonction de comparaison pour compter le nombre d'éléments identiques des deux codes PIN.

```
1  [...]
2  0x805e: e001    b.n     8064              % Saut par-dessus les données
3  0x8060: 82e0    strh    r0, [r4, #22]     % 16 bits de poids faible
4  0x8062: 2402    movs    r4, #2            % 16 bits de poids fort
5  0x8064: 68fb    ldr     r3, [r7, #12]
6  0x8066: 2b01    cmp     r3, #1
7  0x8068: ddea    ble.n   8040  % Si r3<2; comparaison du prochain octet
8  [...]              % Série d'instructions nop ajoutée artificiellement
9  0x829e: 2c02    cmp     r4,#2             % Authentification si r4 == 2
10 0x82a0: d101    bne.n   82a6
11 0x82a2: 2301    movs    r3, #1          % r3=1; le code PIN est valide
12 0x82a4: e000    b.n     82a8
13 0x82a6: 2300    movs    r3, #0        % r3=0; le code PIN est invalide
14 0x82a8: 4618    mov     r0,r3
15 0x82aa: 3710    adds    r7, #16
16 0x82ac: 46bd    mov     sp, r7
17 0x82ae: bc90    pop     {r4, r7}
```

Listing 5.3: Parties du code assembleur ARM de la fonction `VerifyPIN` modifiée

Cette fonction a été analysée face au modèle de faute de saut d'instruction. Le prédicat *Vuln* dans la formule F_{vuln} exprime l'égalité de la valeur de retour à la valeur `0xAA` (valeur associée à une authentification). Comme pour les expériences précédentes, les deux codes PIN ont été symbolisés de façon à couvrir toutes les valeurs possibles des codes PIN et les deux codes PIN sont contraints à être différents.

Parmi toutes les vulnérabilités trouvées par l'analyse, deux d'entre elles correspondent au saut de l'instruction de branchement à l'adresse `0x805e` : à toutes les itérations de la boucle dans la fonction de comparaison. Cette faute force l'exécution des instructions en séquence, *i.e.* des instructions correspondant à la donnée interprétée comme du code. Comme mentionné précédemment l'instruction `movs` (ligne 4) écrit la valeur `2` dans le registre `r4` qui contient le nombre d'octets identiques des deux codes PIN. Le contenu du registre `r4` est ensuite comparé à la valeur 2, le registre contenant la valeur de retour est donc affecté avec la valeur indiquant une authentification alors que les deux codes PIN

sont différents.

Cette expérience montre la capacité de `RobustB` à détecter automatiquement si le placement de données dans la section `.text` par le compilateur introduit des vulnérabilités exploitables avec une attaque en faute, et ce grâce au désassemblage des blocs de données réalisé avec `capstone`. Cette expérience montre un autre cas où une analyse de robustesse effectuée au niveau binaire est nécessaire à la découverte de toutes les vulnérabilités.

5.7 Analyse de code système

FreeRTOS [8] est un système d'exploitation temps réel libre et indépendant. Il est développé par Real Time Engineers Ltd. Dans cette expérience, nous analysons une fonction de l'implémentation FreeRTOS-MPU compilée pour le microcontrôleur Stellaris LM3S811 incluant un processeur ARM Cortex-M3. FreeRTOS-MPU est une version de FreeRTOS qui utilise une unité de protection mémoire (MPU) ainsi que les différents niveaux de privilèges offerts par le processeur Cortex-M3 [9].

Cette expérience vise l'analyse de robustesse d'une fonction du noyau qui permet de charger le contexte de la première tâche utilisateur devant s'exécuter en mode non privilégié. Le code de la fonction du noyau est donnée dans le listing 5.4. Au démarrage du système, toutes les tâches programmées par l'utilisateur sont initialisées et le processeur est en mode privilégié. La fonction `prvRestoreContextOfFirstTask` est appelée juste avant l'exécution de la première tâche utilisateur. Cette fonction prépare le contexte d'exécution de la première tâche utilisateur et change le niveau de privilège du processeur de privilégié au niveau requis par la tâche utilisateur. Le changement du niveau de privilège est effectué par l'écriture de la valeur `0x3` dans le registre spécial `CONTROL` par l'instruction privilégiée `msr` (ligne 13 du listing 5.4). Le bit '1' du registre `CONTROL` indique la pile qui est utilisée (pile OS ou pile processus) et le bit '0' indique le niveau de privilège de l'exécution (donnant accès ou non à l'exécution d'instructions privilégiées et à la manipulation des registres privilégiés). Le saut de l'instruction `msr` ou la modification de la valeur `0x3` pourrait donc donner, à la première tâche, un accès à la pile du système d'exploitation, les droits d'une application privilégiée, ou les deux.

Le prédicat *Vuln* de la formule F_{vuln} exprime l'inégalité des deux bits de poids faible du registre `CONTROL` avec la valeur `0x3`. L'analyse a été effectuée considérant les modèles de

8. FreeRTOS (Free Real Time Operating System) : https ://www.freertos.org/
9. Niveaux de privilèges dans le processeur Cortex-M3 : http ://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/CHDIGFCA.html

```
1  0x12f0: ldr   r0, [pc, #1072]
2  0x12f4: ldr   r0, [r0, #0]
3  0x12f6: ldr   r0, [r0, #0] % Récupère la valeur pointeur de pile système
4  0x12f8: msr   MSP, r0              % Affecte le pointeur de pile système
5  0x12fc: ldr   r3, [pc, #48]
6  0x12fe: ldr   r1, [r3, #0]
7  0x1300: ldr   r0, [r1, #0]  % Récupère l'adresse de la pile de la tâche
8  0x1302: add   r1, r1, #4          % Manipulation des registres de la MPU
9  0x1306: ldr   r2, [pc, #1056]     % Manipulation des registres de la MPU
10 0x130a: ldmia r1!, {r4-r11}       % Manipulation des registres de la MPU
11 0x130e: stmia r2!, {r4-r11}       % Manipulation des registres de la MPU
12 0x1312: ldmia r0!, {r3, r4-r11}      % Dépile les registres temporaires
13 0x1316: msr   CONTROL, r3 % Change le mode d'exécution en non privilégié
14 0x131a: msr   PSP, r0          % Restaure le pointeur de pile de la tâche
15 0x131e: mov   r0, #0
16 0x1322: msr   BASEPRI, r0
17 0x1326: mvn   lr, #2
18 0x132a: bx    lr
```

Listing 5.4: Code assembleur ARM de la fonction `prvRestoreContextOfFirstTask`

faute de saut d'instruction et de corruption de registre. Elle a révélé dix vulnérabilités permettant de donner à une tâche utilisateur des droit privilégiés ou l'accès à la pile du système d'exploitation. L'inspection de ces résultats nous a permis de trouver que huit de ces dix vulnérabilités se trouvent être sur la chaîne def-use (définition-utilisation) du registre `r3` avant que sa valeur ne soit transférée au registre `CONTROL` (ligne 13). Les registres vulnérables sont surlignés dans le listing 5.4. La corruption de l'un de ces registres, conséquence du saut d'une instruction affectant leur valeur, mène à une corruption du bit ou des deux premiers bits de poids faible du registre `CONTROL`. Les vulnérabilités restantes concernent la corruption du registre `r2` après l'exécution de l'instruction ligne 9 et avant l'exécution de l'instruction ligne 11. Cette instruction correspond à une instruction d'écriture en mémoire de plusieurs registres (ici, tous les registres compris entre `r4` et `r11` inclus sont écrits à l'adresse contenue dans le registre `r2`). La corruption du registre `r2` par une valeur contenue dans la plage [`r0`, `r0` + 28], a pour conséquence d'écraser les valeurs qui sont lues ligne 13 et donc, de corrompre la future valeur du registre `r3` qui se propage ensuite au registre `CONTROL`.

Cette expérience montre la capacité de `RobustB` à mettre en lumière des vulnérabilités difficilement visibles à la revue de code : l'analyse de chaîne def-use est commune, mais la prise en compte des problèmes d'alias mémoire (*memory aliasing*) [10], plus difficilement visibles, est simplifiée par le caractère exhaustif de l'analyse.

Les codes systèmes incluent souvent directement du code assembleur rendant leur analyse à bas niveau difficile ; et une analyse automatisée aide le concepteur en lui mettant

10. Memory aliasing : décrit un accès aux mêmes emplacements mémoire à partir de symboles différents

en lumière les instructions sensibles. Cet exemple montre aussi que l'analyse de code système (incluant l'utilisant d'instructions et de registres privilégiés) est possible avec RobustB.

5.8 Performance de RobustB

On s'intéresse dans cette partie à la performance de RobustB et aux éléments qui jouent un rôle dans la durée des analyses de robustesse.

Le tableau 5.8 présente des informations concernant les temps de résolution des expériences présentées dans les sections précédentes. Les colonnes *Analyse* et *Version* indiquent respectivement les codes analysés et leurs versions. La colonne *#fautes* indique le nombre de fautes injectées. La colonne $\overline{\#inst.}$ indique le nombre moyen d'instructions par exécution. La colonne $\#CR$ montre le nombre de chemins de référence. La colonne $\#PR$ indique le nombre de problèmes de robustesse donnés au solveur SMT tandis que $\#PR$ *sat* indique le nombre de ces problèmes dont la satisfiabilité a été vérifiée. La colonne *Acc.* indique le nombre de problèmes de faisabilité donnés au solveur SMT tandis que la colonne *Acc. sat* indique le nombre de ces problèmes dont la satisfiabilité a été vérifiée. La colonne *Tps. total* donne la durée de l'analyse. La colonne *Tps. cumulé* donne la durée des temps de résolution cumulé de tous les problèmes SMT générés pendant l'analyse.

Étant donné que RobustB est, pour la majeure partie de son temps d'exécution, en attente de la fin de la résolution de problèmes SMT, le rapport entre tps. cumulé et tps. total d'une même analyse correspond au degré de parallélisme exhibé par RobustB. La moyenne de leur rapport sur toutes les expériences, dont le temps total est supérieur à 5 minutes, est de 3,2 avec une variance de 1,31. Le degré de parallélisme exhibé par RobustB pour ces expériences est donc compris entre $\sim$2 et $\sim$4.5 (sur une machine contenant 40 cœurs). Les problèmes SMT résolus en parallèle correspondent au problèmes SMT générés pendant l'énumération des chemins de référence des chemins fautés, *i.e.* lorsque la faisabilité des sous-chemins d'un PEFG est testée (cf. section 4.7.1.4).

Le temps de résolution d'une analyse dépend de plusieurs facteurs. Le nombre d'instructions des chemins de référence en est un et, pour plusieurs raisons, il est le plus important. Plus le nombre d'instructions des chemins de référence est élevé, plus le nombre de fautes possibles est élevé ; cela augmente naturellement le nombre de problèmes SMT qui doivent être résolus. Le nombre d'instructions des chemins de référence impacte directement le nombre de variables et d'opérations nécessaires à l'expression des problèmes SMT ; il impacte donc aussi le temps de résolutions des problèmes SMT. Le nombre de chemins

de référence est un autre facteur, et il a en général un impact important sur le temps de résolution de l'analyse : si les chemins de référence ont un nombre semblable d'instructions, alors le temps total (*Tps. total*) de l'analyse correspond au temps de l'analyse d'un chemin de référence multiplié par leur nombre.

Le modèle de faute considéré pour l'analyse joue grandement sur le temps total de l'analyse. En comparant les colonnes *Tps. total* et *#fautes* des analyses des versions de la fonction `memcpy` sécurisée au niveau source face à un saut d'instruction (SI) avec celles de la même fonction face à une corruption de registre (CR), on voit que les temps totaux des premières sont au moins multipliés par 3 par rapport aux secondes. Le nombre de fautes est ∼1.5 fois supérieur dans les analyses considérant un saut d'instruction. Mais, la libération de la valeur d'une variable (suppression de ses contraintes) associée à un registre, qui est réalisée lors de l'application d'une corruption de registre, permet la découverte de plus de chemins faisables qu'avec un saut d'instruction, ce qui a pour effet d'affecter le nombre de problèmes SMT générés (colonnes *#PR + Acc.*). Le nombre de problèmes de faisabilité, témoin du degré d'exploration du PEFG, est 2 à 3 fois plus important dans les analyses considérant une corruption de registre que dans celles considérant un saut d'instruction. Le nombre de problèmes de faisabilité satisfiables (colonne *Acc. sat*) est, quant à lui, 3 à 4 fois plus important dans les analyses considérant une corruption de registre que dans celles considérant un saut d'instruction.

Les dix analyses des versions de la fonction `memcpy` compilées avec LLVM (ensembles B et D) sont, en moyenne, deux fois plus rapides que les dix analyses des versions compilées avec GCC (ensembles A et C). On observe des différences plus grandes encore dans la colonne indiquant les temps cumulés. La différence varie jusqu'à un facteur 4 pour les versions VCa considérant une corruption de registre alors que le nombre de problèmes SMT est similaire. La différence vient donc du temps de résolution des problèmes SMT (*#PR + Acc.*). Pourtant, la quantité d'instructions est quasi identique, tout comme le nombre d'accès mémoire et de branchements conditionnels. Selon nous, cette différence pourrait s'expliquer, d'une part, par la sélection des instructions et l'ordonnancement réalisés par GCC, qui peuvent induire de longues chaînes de dépendance et augmenter la longueur des clauses à résoudre par le solveur SMT ; d'autre part, par la modélisation trop naïve des problèmes SMT qui pourrait, dans certains cas, participer à l'agrandissement des clauses. De la même façon, les temps totaux et cumulés de l'analyse de la fonction $\texttt{VerifyPIN}_5$ (G) sécurisée au niveau source sont bien supérieurs aux versions 0 (E), 4 (F) et 7 (H). La figure 5.4 montre la densité des distributions des temps de résolution des problèmes SMT (en grande majorité de faisabilité) issus des analyses des fonctions $\texttt{VerifyPIN}_4$ (à gauche) et $\texttt{VerifyPIN}_5$ (à droite). On observe qu'un nombre non négligeable de problèmes SMT de l'analyse de $\texttt{VerifyPIN}_5$ sont résolus entre 2.5 et 4 secondes. Le nombre d'instructions moyen (colonne $\overline{\#inst.}$) est

de 149 pour la version $\texttt{VerifyPIN}_4$ et de 124 pour $\texttt{VerifyPIN}_5$; les deux versions ont 15 chemins de référence (colonne $\#CR$) avec 1 chemin de 15 instructions et la variance des 14 restants est quasi nulle. Nous n'avons pas pu déterminer pourquoi ces problèmes prennent plus de temps à être résolus, plus d'investigations sont nécessaires pour comprendre la raison de ces temps de résolution longs. Selon nous, ce ralentissement pourrait être dû à une modélisation trop naïve des problèmes SMT, produisant des expressions d'une forme telle qu'elles seraient difficilement solvables par les solveurs SMT. Cela peut aussi être dû à la présence de longues chaînes de dépendances dans certains codes. Nous avons investigué l'hypothèse d'une modélisation trop naïve des problèmes SMT en considérant une implémentation de l'algorithme AES dont le temps de résolution du problème de faisabilité des chemins nous semblait anormalement long. Nous avons d'abord déterminé manuellement un préfixe de l'unique chemin de référence de l'implémentation dont la faisabilité était solvable par le solveur en un temps raisonnable et tel que l'ajout d'une instruction à ce préfixe augmentait très fortement le temps de l'analyse de faisabilité. En collaboration avec Benjamin Farinier, nous avons pu tester si la cause de ce temps de résolution anormalement longs était liée à une modélisation naïve de la mémoire et de ses accès. En effet, les travaux de M. Farinier visent, entre autres, à simplifier les problèmes SMT par l'optimisation de la manipulation des *arrays* [FDBL18], théorie utilisée par `RobustB` pour la modélisation de la mémoire. L'outil issu de ses travaux, `TFML`, opère une transformation source à source en considérant du code au format SMT-LIB. Les transformations faites par `TFML` sur le probème SMT ont permis de fortement réduire le nombre d'accès en lecture dans le problème SMT : elles passent de 8514 à 1730. Nous avons pu constater que cette réduction du nombre d'accès en lecture n'avait pas eu d'effet significatif sur le temps de résolution du problème SMT de faisabilité du chemin. Nous en avons donc conclu que la modélisation des accès mémoire en lecture n'était pas, dans ce cas là, un facteur impactant fortement le temps de résolution. Il est donc probable que ce soit le problème lui même qui soit trop complexe pour être résolu d'une telle façon ou alors ou que ce soit la modélisation des instructions, au travers de la manipulation de bit-vectors, qui soit le facteur principal impactant les temps de résolution.

La colonne $\#PR$, indiquant le nombre de problèmes de robustesse, de l'analyse de la fonction VerifyPIN_0 (I) sécurisée à la compilation indique la valeur 0. Cela signifie que `RobustB` a pu conclure sur la robustesse de la fonction sans générer de problèmes de robustesse. Ce phénomène s'explique par la structure de la fonction. En effet, cette fonction a deux points de sortie, l'un est emprunté lorsque les codes PIN sont identiques (authentification) et l'autre lorsqu'ils sont différents (non-authentification). Pour une telle structure, les chemins de référence et les chemins fautés ne sont pas déroulés avec le même point d'observation : les premiers sont déroulés avec comme point d'observation le point de sortie de non-authentification tandis que les seconds sont déroulés avec comme point

d'observation le point de sortie d'authentification. La fonction étant robuste à un saut d'instruction, les analyses de faisabilité des chemins fautés n'ont pu atteindre le point d'observation correspondant à une authentification.

TABLE 5.5: Éléments de comparaison de performance des analyses de robustesse. La mention sec. source indique que le code a été sécurisé au niveau source et la mention sec. comp. indique qu'il a été sécurisé à la compilation. La mention Opti. indique le niveau d'optimisation avec lequel les versions ont été compilées. La mention MF est suivie du modèle de faute considéré lors de l'analyse ; RC pour corruption de registre et SI pour saut d'instruction (*) Le nombre de points d'injection et le nombre d'instructions dans l'analyse est anormalement grand par rapport à son temps de résolution ; cela est du aux instructions `nop` ajoutées artificiellement.

Analyse	*Version*	*#fautes*	$\overline{\#inst.}$	*#CR*	*#PR*	*#PR sat*	*Acc.*	*Acc. sat*	*Tps. total*	*Tps. cumulé*	
`memcpy` sec. source. MF : SI	VCa GCC	652	77	8	1120	0	30452	16976	21m	70m	A
	VC1 GCC	636	76	8	2436	52	27248	15580	20m	68m	
	VC2 GCC	604	72	8	2404	81	27216	15548	18m	61m	
	VC3 GCC	620	74	8	1100	6	29580	16600	21m	70m	
	VC4 GCC	660	78	8	1155	0	30468	17111	22m	72m	
	VCa LLVM	648	78	8	1596	0	33420	19454	10m	29m	B
	VC1 LLVM	608	74	8	3184	16	32072	20340	10m	30m	
	VC2 LLVM	572	68	8	2876	46	31092	18754	9m	28m	
	VC3 LLVM	604	72	8	2900	14	31204	18820	10m	29m	
	VC4 LLVM	620	74	8	2900	0	31112	18758	11m	35m	
`memcpy` sec. source. MF : RC	VCa GCC	1016	77	8	3176	26	72020	44368	77m	327m	C
	VC1 GCC	976	76	8	8563	400	63292	41162	74m	320m	
	VC2 GCC	944	72	8	8531	812	63260	41140	55m	215m	
	VC3 GCC	968	74	8	3492	164	71872	44606	65m	268m	
	VC4 GCC	1008	78	8	3632	67	71944	44778	65m	270m	
	VCa LLVM	864	78	8	3796	16	85832	53748	30m	80m	D
	VC1 LLVM	808	74	8	8000	29	82704	56672	38m	95m	
	VC2 LLVM	752	68	8	7968	106	87848	59368	36m	99m	
	VC3 LLVM	800	72	8	8016	62	87920	59440	39m	108m	
	VC4 LLVM	832	74	8	8048	30	87960	59472	36m	95m	
`VerifyPIN` sec. source. Opti. : O0. MF : SI	$\texttt{VerifyPIN}_0$	341	84	4	865	96	5057	3491	10m	27m	E
	$\texttt{VerifyPIN}_4$	2239	149	15	11853	112	136141	83671	114m	197m	F
	$\texttt{VerifyPIN}_5$	1908	124	15	11924	15	152318	94125	907m	6763m	G
	$\texttt{VerifyPIN}_7$	2704	180	15	11362	67	122497	75775	104m	264m	H
`VerifyPIN` sec. source. Opti. : O2. MF : SI	$\texttt{VerifyPIN}_0$	116	29	4	74	54	918	666	1m	2m	
	$\texttt{VerifyPIN}_4$	51	49	1	35	35	429	205	2m	4m	
	$\texttt{VerifyPIN}_5$	53	48	1	48	8	716	389	2m	7m	
	$\texttt{VerifyPIN}_7$	51	50	1	26	24	310	159	30s	1m	
`VerifyPIN` sec. comp. MF : SI	$\texttt{VerifyPIN}_0$	265	66	4	0	0	5238	3165	10m	16m	I
	$\texttt{VerifyPIN}_5$	290	72	4	936	0	9608	5464	16m	24m	
`memcpy` sec. comp. MF : SI, RC	N/A	130	49	1	117	13	572	295	3m	4m	
Data in Codes. MF : SI	N/A	335(*)	335(*)	1	348	7	1330	852	2m	2m	
FreeRTOS. MF : SI, RC	N/A	39	18	1	38	10	39	39	5s	9s	

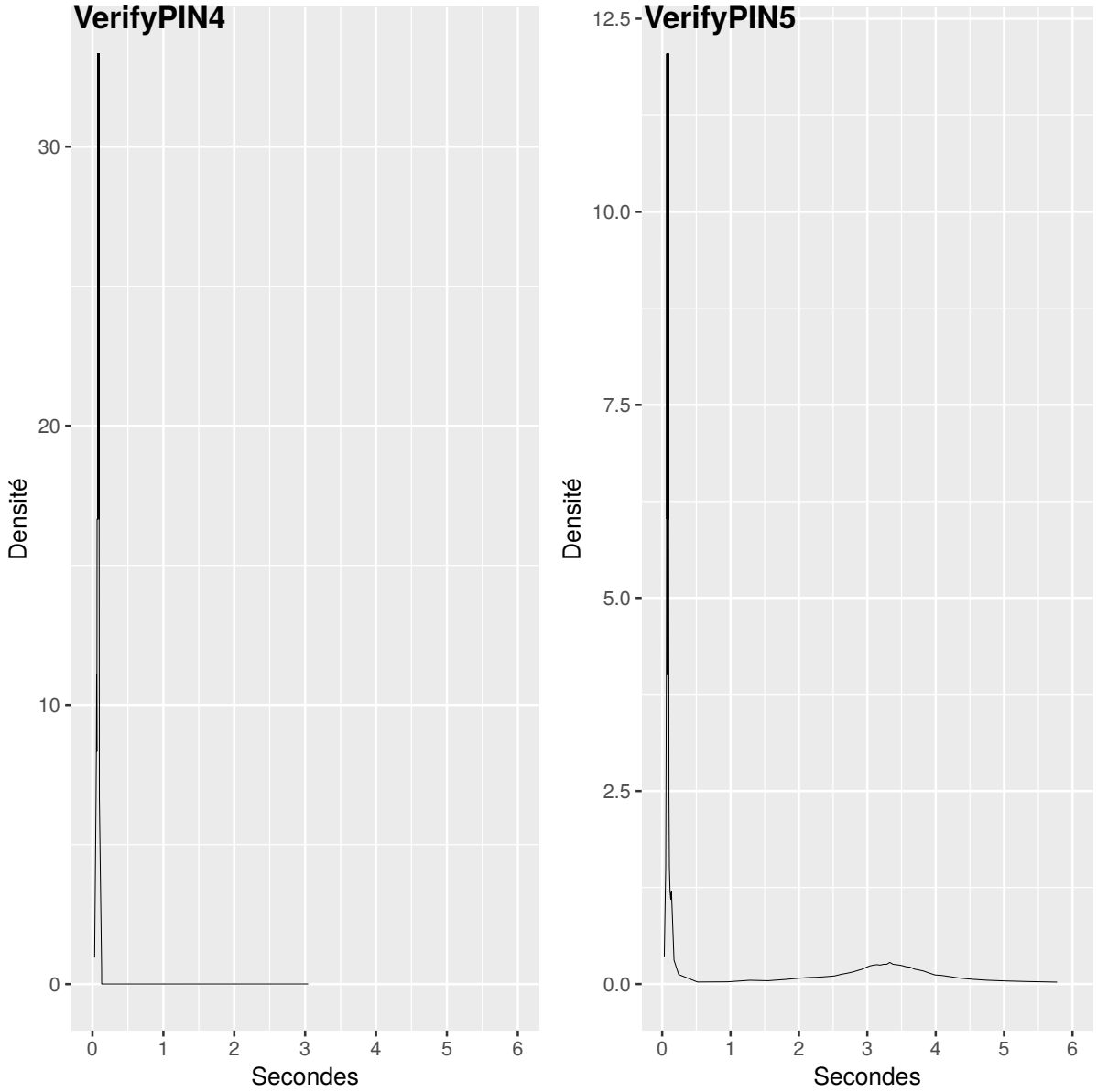

FIGURE 5.4: Distribution des temps de résolution des problèmes SMT des $\texttt{VerifyPIN}_4$ et $\texttt{VerifyPIN}_5$ sécurisées au niveau source et compilées au niveau d'optimisation `O0`

5.9 Conclusion

Dans ce chapitre nous avons illustré plusieurs cas d'utilisation de `RobustB` avec huit études regroupées en six parties. Ces études sont suivies d'une discussion sur la performance de `RobustB` et les éléments l'impactant, vis-à-vis des temps de résolution des analyses présentées dans ce chapitre.

Nous avons d'abord montré comment notre outil peut être utilisé pour analyser les effets de la compilation, et en particulier de ses optimisations sur de multiples versions de la fonction `VerifyPIN`. Nous avons déterminé quelle était la version la plus robuste face à

un saut d'instruction, et les métriques nous ont permis de raisonner sur les effets du niveau d'optimisation `O2` sur les versions protégées de la fonction et sur la version non protégée. Cette expérience a aussi mis en évidence qu'un code protégé optimisé était moins robuste qu'une version non protégée et optimisée du même code. Ce paradoxe est connu [Dur16] et nous avons montré qu'il était exacerbé par la compilation au niveau d'optimisation `O2`.

Nous avons ensuite montré comment `RobustB` peut être utilisé pour vérifier que les protections sont correctement implantées dans l'application lorsqu'elles sont déployées à la compilation. Dans le premier, nous avons pu montrer la robustesse de deux versions de la fonction `VerifyPIN` dont les instructions ont été dupliquées une fois. Le second cas d'étude porte sur l'analyse d'une fonction `memcpy` sur laquelle un schéma de protection de boucle a été automatiquement appliqué à la compilation. Nous avons pu montrer la robustesse de ce code face aux deux modèles de faute considérés par le schéma de protection vis-à-vis de la portée de ce schéma. `RobustB` nous a toutefois permis de trouver une vulnérabilité liée au placement de code, en dehors de la portée du schéma de protection, montrant le poids des détails des transformations du compilateur sur la robustesse du code final.

La troisième partie regroupe les analyses concernant une implémentation d'une fonction `memcpy` contenant des protections et des vérifications implantées au niveau du code source. Nous avons analysé des versions de cette fonction, compilées avec GCC et LLVM au niveau d'optimisation `O2` et face aux modèles de faute de saut d'instruction et de corruption de registre. Nous avons pu mettre en avant l'utilité de la métrique IS pour cibler les instructions vulnérables et comprendre les sources des vulnérabilités dans les codes. Nous avons pu observer l'impact des passes de sélection d'instruction et de l'ordonnancement des instructions sur la robustesse du code binaire final. De plus, `RobustB` a permis de déterminer la redondance de certaines vérifications face à un modèle de faute.

Dans la quatrième partie, nous avons analysé une version de la fonction `VerifyPIN` dans le but de déterminer des chemins d'attaque dont le résultat a servi à la reproduction de la faute avec une attaque au laser. Après la caractérisation du modèle de faute par le CEA LIST, `RobustB` a été en mesure de trouver une vulnérabilité particulièrement critique et de fournir les informations nécessaires à la réalisation de l'attaque.

La cinquième et la sixième partie mettent en avant la capacité de `RobustB` à trouver certaines vulnérabilités liées aux spécificités du code analysé. D'abord, nous avons vu comment notre outil a pu trouver une vulnérabilité dans un code où le compilateur a placé des données d'une manière telle qu'une faute ayant pour effet un saut d'instruction peut permettre leur exécution et, par là, impacter la sécurité du code. Ensuite, nous avons montré un exemple d'utilisation de `RobustB` sur un code issu du système d'exploitation `FreeRTOS` contenant des instructions privilégiées.

Enfin, nous avons discuté des performances de `RobustB` en nous appuyant sur les analyses réalisées dans les six parties. Nous avons mis en avant les éléments impactant la performance mais nous avons aussi relevé des anomalies qui nécessitent plus d'investigations pour les comprendre et potentiellement y remédier.

Les codes analysés (et analysables) par cette méthode, et plus généralement par les méthodes formelles, sont relativement petits. Bien que la taille des codes soit peu discutée dans la littérature nous avons pu observer que l'ordre de grandeur des codes analysables est le même lorsque l'analyse du comportement du code est réalisé avec des méthodes formelles [**?**, Jaf19]. Toutefois, ces méthodes sont en plein essor et, elles permettent la résolution de problèmes de plus en plus important. À l'avenir, ces méthodes devraient permettre l'analyse de codes plus longs et plus complexes.

Chapitre

6 *Conclusion et perspectives*

Les attaques en fautes constituent une menace grandissante vis-à-vis de la sécurité des systèmes embarqués. Un certain nombre de contre-mesures sont mises en place dans ces systèmes, notamment au niveau logiciel, dans le but de garantir la confidentialité et/ou l'intégrité des données qu'ils manipulent. Cette thèse avait pour but de proposer une méthode automatique pour analyser formellement la robustesse de codes binaires en présence de fautes.

Dans un premier temps, nous avons présenté différents moyens disponibles à un attaquant pour réaliser des attaques en fautes, ainsi que les effets possibles des attaques en faute et leur exploitation. Nous avons ensuite présenté les différents moyens de protections prenant la forme de contre-mesures matérielles et logicielles. Nous avons vu les avantages et inconvénients des différents niveaux auxquels peuvent être déployées les contre-mesures logicielles : au niveau du code binaire, au niveau du code source ou sur une représentation intermédiaire lors de la compilation. Quelle que soit la méthode choisie, nous avons vu qu'il est nécessaire de s'assurer de l'efficacité des contre-mesures. En effet, le compilateur peut avoir des effets sur l'intégrité des protections. De plus, l'ajout de protections augmente la surface d'attaque et peut donc introduire de nouvelles vulnérabilités. Nous avons ensuite brièvement présenté le processus de sécurisation d'un code et ses étapes, à savoir : la conception de la contre-mesure, son déploiement et, ce qui nous intéresse ici, l'analyse du code protégé. Nous avons vu que les analyses actuellement utilisées pour analyser un code sont nécessaires mais souffrent de certains défauts : elles sont peu ou pas automatisées, coûteuses et selon la méthode employée elles sont limitées quant à la couverture des comportements possibles et aux types de fautes analysables.

Dans le troisième chapitre, nous avons présenté différentes méthodes d'analyse de robustesse existantes à travers deux principaux critères : la modélisation de l'effet des fautes et les méthodes d'analyses de comportement. Nous avons vu que le niveau auquel la

faute est modélisée ainsi que le moyen utilisé pour représenter son effet sur le code jouent un rôle important dans la capacité de l'analyse à représenter certains effets des fautes. Le moyen utilisé pour modéliser la faute peut, par exemple, limiter l'analyse de robustesse à ne pouvoir faire état que de fautes persistantes. Le niveau de code de modélisation de la faute détermine le niveau de code auquel l'analyse de robustesse opère et nous avons vu que ce niveau impacte fortement la portée de l'analyse : tous les effets observés à bas niveau ne peuvent pas être modélisés à un niveau de code plus haut ; mais l'analyse d'un code haut niveau est plus rapide et peut donc porter de plus grandes portions de code. Nous avons ensuite présenté les différentes méthodes d'analyse de comportement d'un code que nous avons regroupées en deux catégories : méthodes d'analyse concrètes (*e.g.* simulation, exécution native) et symboliques (*e.g.* model-checking, exécution symbolique, résolution SAT/SMT). Nous avons vu que le choix d'une de ces deux méthodes conditionne, lui aussi, la portée de l'analyse. En effet, une analyse réalisée à travers une méthode concrète ne peut que considérer un petit nombre de configurations des valeurs d'entrée du code analysé contre l'intégralité des valeurs dans le cas des analyses réalisées avec une méthode symbolique. Mais, ces dernières passent moins bien à l'échelle que les méthodes concrètes et sont sujettes à des faux positifs. Nous avons ensuite présenté l'état de l'art, où les approches sont organisées par niveau de code analysé : code source, représentation intermédiaire, code assembleur ou binaire et enfin, microarchitecture. Nous avons relevé un manque d'approches automatiques combinant une méthode d'analyse symbolique à bas niveau et permettant la modélisation de fautes transitoires aussi bien sur le code que sur les données.

Dans le quatrième chapitre, nous avons présenté notre approche et son implémentation. Notre approche vise une analyse automatique et contextuelle de la robustesse d'un code binaire combinant à la fois de l'exécution symbolique et des méthodes formelles. L'approche proposée modélise le programme et construit un problème SMT, comparant le comportement de chaque chemin du programme en absence et en présence d'une faute, pour décider si une faute peut mettre en évidence une vulnérabilité. La décision est prise en fonction d'une propriété de sécurité portant sur les valeurs finales d'éléments mémorisant à la fin du programme. Pour limiter le problème des faux positifs, cette comparaison est réalisée vis-à-vis de configurations d'entrée du programme obtenues par une exécution symbolique du programme analysé. Nous proposons ensuite trois métriques permettant de synthétiser les vulnérabilités trouvées lors de l'analyse. Ces métriques, basées sur la probabilité d'exécution d'un chemin et donc sur la probabilité de présence d'une vulnérabilité potentielle, proposent d'attribuer un score de sensibilité à chaque instruction du programme, de donner un score de sensibilité global désignant le nombre moyen de vulnérabilités par chemin et de ramener ce score au nombre d'instructions du programme pour obtenir la densité des vulnérabilités. Dans la partie de ce chapitre dédiée à l'implémentation de l'approche, nous présentons `RobustB`, l'outil qui a été développé tout au long de l'élaboration de l'approche. Nous

présentons les différentes étapes de RobustB, depuis l'exécution symbolique du code binaire jusqu'au traitement des résultats en passant par les différentes analyses de faisabilité et l'analyse de robustesse. Nous détaillons chaque module composant RobustB ainsi que leurs interactions.

Le cinquième chapitre est consacré aux expériences réalisées avec RobustB. Elles sont organisées en six parties où les quatre premières parties proposent de montrer différents usages de RobustB et les deux restantes montrent la capacité de l'outil à rendre compte de spécificités présentes dans certains codes. Dans la première partie, nous avons analysé 4 versions de la fonction VerifyPIN, dont 3 sont protégées. L'analyse a montré l'impact des passes d'optimisation du compilateur sur les protections lorsqu'elles sont insérées au niveau du code source, mais aussi que le paradoxe de la surface d'attaque est exacerbé lorsque les codes sont optimisés. Dans la seconde partie, nous avons pu montrer la robustesse de deux codes protégés à la compilation et donc le bon déploiement des deux schémas de protection par le compilateur. L'analyse de l'un des codes a révélé une vulnérabilité se trouvant en dehors de la portée du schéma de protection, tirant parti d'une information visible uniquement au niveau de code assembleur ou binaire : le placement de code. Dans la troisième partie, nous avons étudié une implémentation sécurisée de la fonction memcpy vis-à-vis de deux compilateurs et de deux modèles de faute. Dans cette expérience, nous avons comparé la robustesse de différentes versions de cette fonction dans le but de déterminer la redondance de protections et ainsi déterminer la version minimale et robuste. L'expérience présentée dans la quatrième partie a été réalisée en collaboration avec les partenaires du projet ANR PROSECCO (2015-2019), dans lequel s'inscrit cette thèse. Nous avons analysé une version protégée à la compilation de la fonction VerifyPIN dans le but de rechercher des chemins d'attaques. Nous avons pu trouver une faute pour laquelle la fonction est vulnérable, quelle que soit la configuration d'entrée. La cinquième et la sixième partie concernent moins un usage de notre outil que sa capacité à prendre en compte des spécificités du code. Nous montrons d'abord que RobustB est capable d'interpréter des données lorsqu'elles sont placées au milieu du code analysé et qui sont donc potentiellement exécutable lorsque que le code est soumis à une faute. Ensuite, nous montrons la capacité de RobustB à rendre compte de la robustesse d'un code contenant des instructions privilégiées à travers l'analyse d'une fonction issue du noyau du système d'exploitation FreeRTOS.

6.1 Perspectives

Nous avons organisé les perspectives de cette thèse en deux parties : celles qui nous semblent réalisables à moyen terme puis les perspectives à long terme.

6.1.1 Moyen terme

Certaines expériences ont montré des anomalies dans la performance de `RobustB` qui correspondent à des temps de résolution de problèmes SMT anormaux. Nous n'avons pas pu trouver la source de ce problème mais nous avons toutefois émis deux hypothèses. Dans la première hypothèse, c'est la forme du problème, et donc la forme du code dont il est tiré, qui produirait une situation difficilement solvable par un solveur SMT. La confirmation de cette hypothèse montrerait alors les limites de l'approche quant à l'analyse de codes présentant de telles formes. Dans la seconde hypothèse, nous pensons que ces anomalies pourraient être le symptôme d'un problème plus général, à savoir, une modélisation des problèmes SMT trop naïve qui augmenterait globalement les temps de résolution des problèmes SMT et, dans certains cas, provoquerait des situations difficilement solvables par un solveur SMT. Si cette hypothèse est confirmée, alors il devrait être possible de produire une autre modélisation des problèmes SMT prenant en compte ce facteur et cela augmenterait alors la complexité et la taille du code analyse par notre approche.

Les modèles de faute dont l'effet sur le code assembleur est dépendant d'éléments architecturaux invisibles du logiciel peuvent être difficiles à représenter au niveau du code binaire. Plus exactement, il peut être difficile de déterminer les effets et les points d'injection de ces modèles de faute. J. Laurent *et al.* montrent dans [LBD+19] que les effets de fautes réalisées sur des éléments d'optimisations du pipeline du processeur dépendent fortement du contexte du code exécuté. Il est nécessaire d'avoir accès à un modèle de l'architecture pour pouvoir déterminer les effets et les points d'injection d'une attaque sur le code impactant des éléments architecturaux. Toutefois, faire apparaître des éléments architecturaux dans la modélisation des problèmes SMT générés est difficilement envisageable sans que l'analyse ne soit considérablement plus lente. Une possibilité pourrait être de simuler le passage des instructions dans un modèle d'architecture pour déduire l'effet des fautes sur les instructions lorsqu'elles visent un élément de l'architecture. Le problème est donc de déterminer le niveau de précision du modèle de l'architecture de façon à ce que sa simulation ne soit pas trop lourde, et, en même temps, de parvenir à déduire l'effet d'un modèle de faute, définit au niveau de l'architecture, sur les instructions.

6.1.2 Long terme

Un solveur SMT est capable de donner la configuration concrète des variables pour laquelle il a trouvé que le problème SMT est satisfiable, *i.e.* il est possible de connaître les valeurs des variables pour lesquelles une vulnérabilité a été trouvée. Cette configuration correspond à la première configuration que le solveur SMT a rencontré qui permet la satisfiabilité de la formule. Or, pour un même problème de robustesse, il peut exister plusieurs configurations concrètes des variables qu'il serait intéressant de pouvoir extraire. Par exemple, lors d'une l'analyse de robustesse d'une faute de type corruption de registre mettant en évidence une vulnérabilité, le solveur SMT donne une valeur de la corruption du registre. Connaître l'intégralité des valeurs de corruption mettant en évidence une vulnérabilité permettrait de mieux comprendre la sensibilité du code analysé et permettrait donc d'agir de façon plus pertinente pour le protéger. Une solution naïve pourrait être de relancer l'analyse en interdisant les valeurs pour lesquelles le solveur a déjà trouvé une vulnérabilité jusqu'à ce qu'il n'y en ait plus. Cette solution n'est pas envisageable dans le cas général étant donné qu'elle implique la génération d'autant de problèmes SMT qu'il existe de valeurs de corruption mettant en évidence une vulnérabilité. Cette solution pourrait être accélérée par l'utilisation d'une technique permettant d'extrapoler les valeurs possibles du registre corrompu en fonction des valeurs finales des éléments mémorisant dans le chemin fauté. Une technique pouvant être utilisée pour extrapoler les valeurs est la *backward analysis*, ou analyse arrière. Cette technique analyse un programme comme s'il s'exécutait à l'envers, depuis la sortie du programme vers ses entrées. Cela pourrait par exemple permettre de déduire un sous-ensembles de valeurs de corruptions menant à une vulnérabilité. Il s'agit toutefois d'un problème compliqué difficilement applicable dans le cas général.

Finalement, une question restant ouverte est : quelles stratégies utiliser pour analyser la robustesse de l'intégralité d'un programme binaire conséquent. Nous avons vu que les méthodes symboliques sont, dans le cas général, les seules capables de conclure sur la robustesse d'un code binaire en présence de fautes, mais elles sont aussi limitées quant à la taille du code analysable. Les analyses hors contexte issues de ces méthodes sont, a priori, composables : si la robustesse est garantie indépendamment sur deux fonctionnalités du code alors elle est aussi garantie sur une séquence où ces deux fonctionnalités se suivent directement (ici, le terme fonctionnalité désigne une séquence d'instructions sur laquelle une propriété peut être exprimée). Un découpage du programme en petites fonctionnalités suffirait donc à ce que celui-ci soit analysable. Toutefois, certaines fonctionnalités ne peuvent pas être découpées. Il est par exemple difficile d'exprimer une propriété pertinente entre deux tours de l'algorithme AES. La combinaison avec une autre approche semble donc indispensable, au moins pour donner une réponse, même partielle, sur la robustesse

de telles fonctionnalités. Un autre problème est celui d'une fonctionnalité f appelant une fonctionnalité g et où seule la fonctionnalité g est assez petite pour être analysée. Si la fonctionnalité g est prouvée robuste, l'analyse de f pourrait être possible par l'utilisation d'une méthode capable d'abstraire le comportement de g (en l'approximant) lors de l'analyse de f.

Des attaques incluant de multiples fautes ont pu être réalisées [KQ07, TK10] et, si ce n'est pas déjà le cas, il est probable qu'elles constituent, à l'avenir, une importante menace à la sécurité des systèmes. L'approche proposée dans cette thèse n'interdit pas les fautes multiples, mais analyser toutes les combinaisons, même pour deux fautes, n'est pas envisageable tant cela augmenterait la durée de l'analyse. Toutefois, les attaquants capables de produire de multiples fautes ont le même problème et doivent donc faire un choix quant à la combinaison de fautes qu'ils souhaitent réaliser. Donc, une analyse de robustesse d'un code soumis à des fautes multiples, définissant à priori un sous-ensemble de combinaisons de fautes, est souhaitable. Néanmoins, la robustesse effective du code dépendra des sous-ensembles choisis et de la capacité de l'attaquant à trouver des combinaisons mettant en évidence une vulnérabilité qui n'aurait pas été envisagée. Un scénario d'attaque possible, proposé par L. Dureuil, dans sa thèse [Dur16], est de réaliser une première faute dont le but est d'impacter directement la sécurité du programme puis de réaliser la deuxième pour désactiver une vérification censée détecter la première. La réalisation d'une analyse robustesse, pouvant trouver des vulnérabilités mettant en cause deux régions de codes disjointes, nécessite de combiner la méthode proposée ici avec une autre (comme de l'exécution symbolique) capable de fournir un contexte fauté au point d'entrée de la seconde région. Toutefois, la technologie progressant, il est possible qu'à l'avenir, les moyens permettant les injections de fautes soient de moins en moins coûteux et de plus en plus performants. Cela donnerait la possibilité à de plus en plus d'attaquants de réaliser de plus en plus de fautes lors d'une même exécution. Dans ce cas, la combinatoire sera telle qu'il sera nécessaire de limiter l'analyse à un nombre de combinaisons de fautes réduites. La façon de déterminer à priori les combinaisons pertinentes est un problème ouvert.

Pour conclure, de nombreuses perspectives et pistes sont encore à explorer. Il nous semble que les travaux futures devront être capables de rendre compte de fautes impliquant des éléments de l'architecture et de multiples fautes. Actuellement, une combinaison d'analyses à haut et bas niveau, indépendamment des techniques utilisée, semble être la solution la plus complète à l'analyse de robustesse d'un programme.

Annexes

Annexe

A Fichier SMT de l'exemple présenté au chapitre 4

Cette annexe a pour but de présenter le contenu des problèmes SMT tels qu'ils sont donnés au solveur SMT. Nous réutilisons l'exemple avec lequel nous avons illustré l'analyse de robustesse au chapitre 4 (cf. 4.7.3).

Le listing A.1 présente le code C correspondant à l'exemple. Les lignes 6 à 8 assignent les valeurs du contexte symbolique aux registres `r0`, `r2` et `r3`. Les lignes 10 à 17 correspondent au code à analyser. Le CFG et le placement des blocs associés au code assembleur sont montrés dans la figure A.1.

```
1  int main(void)
2  {
3      register int val asm("r0");
4      register int r2  asm("r2");
5      register int res asm("r3");
6      val = 0;
7      res = 3;
8      r2 = 0;
9      asm volatile(
10   "        b      lbl             \n"
11   "        nop                    \n"
12   "lbl :   cmp    r0,      #0     \n"
13   "        beq    edest           \n"
14   "        movs   r2,      #0     \n"
15   "        b      fdest           \n"
16   "edest : movs   r2,      #2     \n"
17   "fdest : add    r3,      r2     \n"
18       :  "+r"(val),
19          "=r"(r2),
20          "+r"(res)
21       :
22       :
23       );
24
25     return res;
26 }
```

Listing A.1: code C

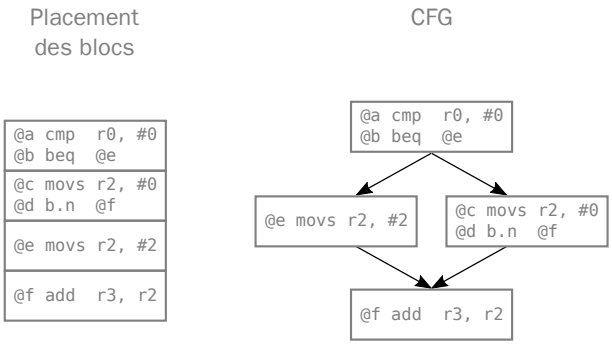

FIGURE A.1: Placement des blocs et CFG du code assembleur

Nous avons compilé ce code C avec les options de compilation `-mcpu=cortex-m4` et `-mthumb` puis désassemblé le binaire résultant avec l'option `-Mforce-thumb`. Le listing A.2 montre le code assembleur de la fonction `main` issu du désassemblage du binaire. Les lignes surlignées (lignes 7 à 20 inclues) correspondent au code utilisé dans l'exemple. L'adresse `812e` est le point d'entrée de la région de code et l'adresse `8138` correspond au point de sortie et au point d'observation.

```
1  00008120 <main>:
2      8120:       b480            push    {r7}
3      8122:       af00            add     r7, sp, #0
4      8124:       2000            movs    r0, #0
5      8126:       2303            movs    r3, #3
6      8128:       2200            movs    r2, #0
7      812a:       e000            b.n     812e <lbl>
8      812c:       bf00            nop
9
10 0000812e <lbl>:
11     812e:       2800            cmp     r0, #0
12     8130:       d001            beq.n   8136 <edest>
13     8132:       2200            movs    r2, #0
14     8134:       e000            b.n     8138 <fdest>
15
16 00008136 <edest>:
17     8136:       2202            movs    r2, #2
18
19 00008138 <fdest>:
20     8138:       4413            add     r3, r2
21     813a:       4618            mov     r0, r3
22     813c:       46bd            mov     sp, r7
23     813e:       bc80            pop     {r7}
24     8140:       4770            bx      lr
```

Listing A.2: Code assembleur issu de la compilation du code C de l'exemple

Nous rappelons ci-dessous la formule F_{vuln} utilisée pour l'analyse :

$$F_{vuln} : \ Fcontexte \wedge Feas(\varphi_r, ctx) \wedge \Big(\bigvee_{\varphi_f \in \varphi_f{}^k_m} Feas(\varphi_f, ctx) \Big) \wedge (r3^{\varphi r}_{V_f} \neq r3^{\varphi f}_{V_f})$$

L'analyse a été réalisée contre le modèle de faute de saut d'instruction visant les instructions aux adresses : `812e`, `8130`, `8136` et `8138`. La figure A.2 montre les résultats de l'analyse. La colonne *Ref Path* indique le numéro du chemin de référence (il n'y en a qu'un pour cette analyse). La colonne *Descr* code le numéro du chemin de référence (rp{numéro}), le numéro de la faute (f{numéro}), le numéro du chemin fauté (fp{numéro})

```
Ref Path  Descr          Faulted inst   Address(occ)  Fault  Result  Verif time

0         rp0_f0_fp0_c0    cmp r0, #0    812e(0)    skip   unsat   0m0.012s
0         rp0_f0_fp1_c0    cmp r0, #0    812e(0)    skip   sat     0m0.023s
0         rp0_f1_fp0_c0    beq #0x8137   8130(0)    skip   sat     0m0.017s
0         rp0_f2_fp0_c0    movs r2, #2   8136(0)    skip   sat     0m0.010s
0         rp0_f3_fp0_c0    add r3, r2    8138(1)    skip   sat     0m0.017s
```

FIGURE A.2: Présentation des résultats de l'analyse

et le numéro du contexte symbolique (c{numéro}). La colonne *Faulted inst* indique l'adresse de l'instruction et son occurrence est montrée entre parenthèse. La colonne *Fault* indique le type de faute, *skip* pour instruction skip. La colonne *Result* donne le résultat de la résolution SMT. Enfin, la colonne *Verif time* donne la durée de la résolution SMT.

Les résultats indiquent que toutes les instructions sont vulnérables et que la faute numéro 0 (sur l'instruction `cmp r0, #0`) permet l'exécution du chemin fauté *fp0* ou *fp1* selon la valeur du drapeau `Z` (cf. 4.7.3).

Dans la suite, nous nous intéressons à la composition et à l'organisation des problème SMT de robustesse. Nous prenons comme exemple le problème associé à la faute `rp0_f0_fp1_c0`.

Les problèmes de robustesse sont organisés en 9 parties :

— les fonctions A.3

— la déclaration des variables du chemin de référence A.4 et du chemin fauté A.5

— le contexte symbolique et l'initialisation des chemins A.6

— le chemin de référence A.7

— le chemin fauté A.8

— la vérification de la propriété A.9

Le code présenté dans les parties suivantes correspond à la syntaxe du standard SMT-LIB. L'écriture des expressions est sous forme de notation polonaise :

$$a * (b + c) \rightarrow (* \ a \ (+ \ b \ c))$$

Le listing A.3 montre la fonction `add` qui modélise un sous ensemble de comportements de l'instruction assembleur du même nom. Cette fonction ajoute le contenu de deux registres source pour l'assigner à un registre destination. Elle met aussi à jour les drapeaux `N` (indiquant un résultat négatif), `V` (indiquant un dépassement de capacité) et `Z` (indiquant

que le résultat de l'opération vaut 0). Nous avons modélisé un grand nombre de fonctions qui modélisent toutes des sous ensembles des comportements d'instructions existantes. Ces fonctions sont ensuite utilisées dans les formules qui modélisent un chemin.

Les listings A.4 et A.5 montrent la déclaration des variables du chemin de référence et du chemin fauté. Un ensemble de variables est déclaré pour chaque instruction ainsi que deux autres ensembles : le premier représente l'état du programme avant l'exécution de la première instruction et le second est l'ensemble final sur lequel est vérifié la propriété.

Les déclarations des variables sont réalisées à partir d'une expression ayant la forme suivante :

```
(declare-fun nom_de_la_variable () (type_de_la_variable))
```

La mention `_ BitVec {nombre}` indique que la variable est un vecteur de bits de la taille du nombre. La mention `(Array (_ BitVec 30) (_ BitVec 32))` indique que la variable est un `Array` (qui peut être vu comme un dictionnaire) pouvant être adressé sur 30 bits et contenant des variables de 32 bits. Les variables de ce type sont utilisées pour représenter la mémoire.

Les noms variables suivent le format suivant :

```
prefix_s{numéro_de_l'ensemble ou f}_chemin
```

La mention `sf` indique que la variable appartient à l'ensemble final.

Il existe quatre préfixes différents :

— `w_r{numéro du registre}` pour les variables qui représentent un registre

— `f` pour les variables qui représentent des drapeaux

— `m` pour les variables qui représentent la mémoire

— `f_w_r{numéro du registre}` pour les variables qui représentent des drapeaux internes à un registre (ces variables ne constituent pas un élément de l'architecture). Les drapeaux internes sont toujours mis à jour dans les fonctions mais il ne sont recopiés dans les drapeaux globaux (correspondant aux drapeaux de l'architecture) que lorsque l'instruction le requiert.

— `w_t{numéro du temporaire}` pour des variables temporaires utilisées pour contenir des calculs intermédiaires (ces variables ne représentent pas un élément de l'architecture).

```
[...] +318 lignes ------------------------------------------

; add
(define-fun add (
        // arguments de la fonction
        (w_dst  (_ BitVec 32)) // registre de destination
        (f_dst  (_ BitVec  4)) // drapeaux pour une éventuelle mise à jour
        (w_src0 (_ BitVec 32)) // registre source 0
        (w_src1 (_ BitVec 32)) // registre source 1
        )
        Bool                   // type de retour de la fonction
  (and    true
    (=
      (concat (intern_get_c f_dst) w_dst)     // concaténation du bit de retenue
      (bvadd                                  // ajout de src0 et src1
        (concat ((_ extract 0 0) #x0) w_src0) // concaténation de 0b0 avec src0
        (concat ((_ extract 0 0) #x0) w_src1) // concaténation de 0b0 avec src0
      )
    )
    (=
      (intern_get_n f_dst) // mise à jour n (résultat négatif)
      ((_ extract 31 31) w_dst)
    )
    (=
      (intern_get_v f_dst) // mise à jour flag v (dépassement)
      (bvor
        (bvand
          (bvnot ((_ extract 31 31) w_dst))
          (bvand (bvnot ((_ extract 31 31) w_src0))
          (       (_ extract 31 31) w_src1))
        )
        (bvand
          (       (_ extract 31 31) w_dst)
          (bvand (       (_ extract 31 31) w_src0)
          (bvnot ((_ extract 31 31) w_src1)))
        )
      )
    )
    (=
      (intern_get_z f_dst) // mise à jour flag z (zéro)
        (bvand (bvnot ((_ extract 0 0) w_dst))
        (bvand (bvnot ((_ extract 1 1) w_dst))
[...] +28 lignes ---------------------------------------------
          (bvand (bvnot ((_ extract 30 30) w_dst))
                 (bvnot ((_ extract 31 31) w_dst))
          ))))))))))))))))))))))))))))))
      )
    )
  )
)

[...] +943 lignes ------------------------------------------
```

Listing A.3: Fonction modélisant l'instruction `add`

```
; Variables declaration for reference trace
(declare-fun    w_r00_s0000_reference () (_ BitVec 32))
(declare-fun f_w_r00_s0000_reference () (_ BitVec 4))
(declare-fun    w_r01_s0000_reference () (_ BitVec 32))
(declare-fun f_w_r01_s0000_reference () (_ BitVec 4))
[...] +40  lignes ------------------------------------------------------------
(declare-fun    w_r22_s0000_reference () (_ BitVec 32))
(declare-fun f_w_r22_s0000_reference () (_ BitVec 4))
(declare-fun    w_t00_s0000_reference () (_ BitVec 32))
(declare-fun f_w_t00_s0000_reference () (_ BitVec 4))
[...] +4   lignes ------------------------------------------------------------
(declare-fun    w_t03_s0000_reference () (_ BitVec 32))
(declare-fun f_w_t03_s0000_reference () (_ BitVec 4))
(declare-fun       f_s0000_reference () (_ BitVec 4))
(declare-fun    w_r00_s0001_reference () (_ BitVec 32))
(declare-fun f_w_r00_s0001_reference () (_ BitVec 4))
[...] +217 lignes ------------------------------------------------------------
(declare-fun       f_s0004_reference () (_ BitVec 4))
(declare-fun       m_s0000_reference () (Array (_ BitVec 30) (_ BitVec 32)))
(declare-fun          m_sf_reference () (Array (_ BitVec 30) (_ BitVec 32)))
(declare-fun      w_r00_sf_reference () (_ BitVec 32))
(declare-fun      w_r01_sf_reference () (_ BitVec 32))
[...] +20  lignes ------------------------------------------------------------
(declare-fun      w_r22_sf_reference () (_ BitVec 32))
(declare-fun          f_sf_reference () (_ BitVec 4))
```

Listing A.4: Déclaration des variables du chemin de référence

```
; Variables declaration for faulty trace
(declare-fun    w_r00_s0000_faulty () (_ BitVec 32))
(declare-fun f_w_r00_s0000_faulty () (_ BitVec 4))
(declare-fun    w_r01_s0000_faulty () (_ BitVec 32))
(declare-fun f_w_r01_s0000_faulty () (_ BitVec 4))
[...] +40  lignes ------------------------------------------------------------
(declare-fun    w_r22_s0000_faulty () (_ BitVec 32))
(declare-fun f_w_r22_s0000_faulty () (_ BitVec 4))
(declare-fun    w_t00_s0000_faulty () (_ BitVec 32))
(declare-fun f_w_t00_s0000_faulty () (_ BitVec 4))
[...] +4   lignes ------------------------------------------------------------
(declare-fun    w_t03_s0000_faulty () (_ BitVec 32))
(declare-fun f_w_t03_s0000_faulty () (_ BitVec 4))
(declare-fun       f_s0000_faulty () (_ BitVec 4))
(declare-fun    w_r00_s0001_faulty () (_ BitVec 32))
(declare-fun f_w_r00_s0001_faulty () (_ BitVec 4))
[...] +217 lignes ------------------------------------------------------------
(declare-fun       f_s0004_faulty () (_ BitVec 4))
(declare-fun       m_s0000_faulty () (Array (_ BitVec 30) (_ BitVec 32)))
(declare-fun          m_sf_faulty () (Array (_ BitVec 30) (_ BitVec 32)))
(declare-fun      w_r00_sf_faulty () (_ BitVec 32))
(declare-fun      w_r01_sf_faulty () (_ BitVec 32))
[...] +20  lignes ------------------------------------------------------------
(declare-fun      w_r22_sf_faulty () (_ BitVec 32))
(declare-fun          f_sf_faulty () (_ BitVec 4))
```

Listing A.5: Déclaration des variables du chemin fauté

```
; init registers declaration
(declare-fun ai_regs_0 () (_ BitVec 32))
(declare-fun ai_regs_1 () (_ BitVec 32))
[...] +13 lignes ------------------------------------
(declare-fun ai_regs_15 () (_ BitVec 32))
; ai concret regs init
(assert (and    true
                (= ai_regs_0 #x00000000)   // r0 = 0
                (= ai_regs_2 #x00000000)   // r2 = 0
                (= ai_regs_3 #x00000003)   // r3 = 3
                (= ai_regs_7 #x7fefffc)    // fp
                (= ai_regs_13 #x7fefffc)   // sp
                (= ai_regs_15 #x00008130)  // pc = 0x8130
        )
)
; mem declaration
(declare-fun ai_mem_word_7fefffc () (_ BitVec 32))
; init
// contraintes des valeurs des registres et de la mémoire du chemin de référence
(assert (and    true
                (= w_r00_s0000_reference ai_regs_0)
[...] +14 lignes ------------------------------------
                (= w_r15_s0000_reference ai_regs_15)
                // assignation mémoire
                (=
                  (select m_s0000_reference
                          ((_ extract 31 2) (_ bv2147418108 32))
                  )
                  ai_mem_word_7fefffc
                )
        )
)

// contraintes de chaque élément du fauté avec son homologue dans le chemin de référence
(assert (and    true
                (= w_r00_s0000_reference w_r00_s0000_faulty0)
[...] +21 lignes ------------------------------------
                (= w_r22_s0000_reference w_r22_s0000_faulty0)
                (=
                  (select m_s0000_reference
                    ((_ extract 31 2) (_ bv2147418108 32))
                  )
                  (select m_s0000_faulty0
                    ((_ extract 31 2) (_ bv2147418108 32))
                  )
                )
        )
)
```

Listing A.6: Contexte symbolique et initialisation des chemins

Le listing A.6 montre comment l'initialisation des chemins par un contexte symbolique est implémenté. Les variables initiales (variables de l'ensemble `s0000`) des chemins sont contraintes par les variables du contexte symbolique (préfixées par la mention `ai`) :

— `ai_regs_{numéro}` indique que la variable représente le registre du numéro correspondant

— `ai_mem_word_{adresse}` indique que la variable représente le mot mémoire associé à l'adresse renseignée

L'opération `select` effectuée sur un `Array` permet de sélectionner une case du tableau.

Le listing A.7 montre la modélisation du chemin de référence et le listing A.8 montre

celle du chemin fauté. Nous décrivons, ci-dessous, les opérations qui sont effectuées dans ces chemins :

— `setval variable valeur` contraint une variable à la valeur indiquée.

— `sub variable_destination drapeaux_variable_destination variable_source_0 variable_source_1` est un appel à la fonction `sub` réalisant la soustraction de `variable_source_0` avec `variable_source_1` pour mettre le résultat dans variable_destination. Elle met aussi à jour les drapeaux de la variable de destination.

— `copyflags drapeaux_destination drapeaux_source` transfère l'expression de drapeaux_source vers drapeaux_destination. Elle est utilisée pour mettre à jour les drapeaux globaux lorsque l'instruction le requiert.

— `intern_get_{drapeaux}` extrait la valeur d'un drapeau

— `mov variable_destination drapeaux_variable_destination variable_source drapeaux_variable_source drapeaux` est un appel à la fonction `mov` modélisant l'instruction `mov` qui transfert la valeur d'un registre source vers un registre destination

Les variables `referencepath` et `faulty1path` contiennent le résultat de la conjonction des opérations du chemin de référence et du chemin fauté respectivement. La satisfiabilité d'une de ces variables représente donc la faisabilité du chemin correspondant.

Le listing A.9 montre la formule par laquelle la propriété est vérifiée. Cette formule peut être réécrite comme suit :

$$ref \wedge \overline{(flt \rightarrow (r03_{sf}^{faulty1} = r03_{sf}^{reference}))} = ref \wedge flt \wedge (r03_{sf}^{faulty1} \neq r03_{sf}^{reference})$$

avec *ref* désignant `referencepath`, *flt* désignant `faulty1path` et $registre_{sf}^{chemin}$ désignant un registre dans l'ensemble final (`sf`) d'un chemin. Cette formule est donc satisfiable si le chemin de référence et le chemin fauté sont faisables et que les registres `r3`, des ensembles finaux des deux chemins, sont différents.

```
(define-fun referencepath () Bool
  (and  true

; @812e: cmp r0, #0
    (setval w_t00_s0001_reference #x00000000)
    (sub
        w_t01_s0001_reference f_w_t01_s0001_reference
        w_r00_s0000_reference w_t00_s0001_reference
    )
    (copyflags f_s0001_reference f_w_t01_s0001_reference)
    (setval w_r15_s0001_reference #x00008134)

; @8130: beq #0x8137
    (setval w_r15_s0002_reference #x00008138)
    (= (intern_get_z f_s0001_reference) (_ bv1 1))

; @8136: movs r2, #2
    (setval w_t00_s0003_reference #x00000002)
    (mov
         w_r02_s0003_reference f_w_r02_s0003_reference
         w_t00_s0003_reference f_w_t00_s0003_reference
         f_s0001_reference
    )
    (copyflags f_s0003_reference f_w_r02_s0003_reference)
    (setval w_r15_s0003_reference #x0000813c)

; @8138: add r3, r2
    (add
        w_r03_s0004_reference f_w_r03_s0004_reference
        w_r03_s0000_reference w_r02_s0003_reference
    )

// contraintes de l'ensemble final
    (= m_sf_reference m_s0000_reference)
    (= w_r00_sf_reference w_r00_s0000_reference)
    (= w_r01_sf_reference w_r01_s0000_reference)
[...] +20 lignes ----------------------------------------
    (= w_r22_sf_reference w_r22_s0000_reference)
    (= f_sf_reference f_s0003_reference)
  )
)
```

Listing A.7: Chemin de référence

```
(define-fun faulty1path () Bool
  (and  true

; @812e: SKIP
    (setval w_r15_s0001_faulty1 #x00008134)

; @8130: beq #0x8137
    (setval w_r15_s0002_faulty1 #x00008134)
    (= (intern_get_z f_s0000_faulty1) (_ bv0 1))

; @8132: movs r2, #0
    (setval w_t00_s0003_faulty1 #x00000000)
    (mov
      w_r02_s0003_faulty1 f_w_r02_s0003_faulty1
      w_t00_s0003_faulty1 f_w_t00_s0003_faulty1
      f_s0000_faulty1
    )
    (copyflags f_s0003_faulty1 f_w_r02_s0003_faulty1)
    (setval w_r15_s0003_faulty1 #x00008138)

; @8134: b #0x8139
    (setval w_r15_s0004_faulty1 #x0000813c)
    true

; @8138: add r3, r2
    (add
      w_r03_s0005_faulty1 f_w_r03_s0005_faulty1
      w_r03_s0000_faulty1 w_r02_s0003_faulty1
    )
// contraintes de l'ensemble final
    (= finished_path #x00000001)
    (= m_sf_faulty1 m_s0000_faulty1)
    (= w_r00_sf_faulty1 w_r00_s0000_faulty1)
    (= w_r01_sf_faulty1 w_r01_s0000_faulty1)
[...] +20 lignes ----------------------------------
    (= w_r22_sf_faulty1 w_r22_s0000_faulty1)
    (= f_sf_faulty1 f_s0003_faulty1)
  )
)
```

Listing A.8: Chemin fauté

```
(assert
  (and
    referencepath
    (not
      (=>
        faulty1path
        (=
          w_r03_sf_faulty1
          w_r03_sf_reference
        )
      )
    )
  )
)
```

Listing A.9: Modélisation de la formule vérifiant la propriété

Annexe

B Passage de la propriété depuis un fichier source écrit en C

Cette annexe présente un exemple de propriété et de contrainte initiale exprimées au niveau d'un code source C ainsi que leur transcription dans un problème SMT.

Le listing B.1 montre le code source C d'une fonction `main` appelant une autre fonction d'authentification par code PIN (la fonction `verifyPIN_2`). Cette dernière compare deux codes PIN, contenus dans les variables `g_userPin` et `g_cardPin`) et assigne la valeur `0xAA` à la variable `g_authenticated` si les codes PIN sont égaux et `0x55` s'ils sont différents. Les codes PIN sont des tableaux de type UBYTE (1 octet non signé) et `PIN_SIZE` vaut 4.

La propriété (ligne 27) est exprimée sous la forme d'une annotation (elle n'a donc pas d'impact sur le code binaire généré à la compilation) et exprime la non égalité de la variable `g_authenticated` avec la valeur `0xAA`. La contrainte (lignes 18-22) exprime qu'au moins un des chiffres des codes PIN doit être différent. Nous cherchons donc une vulnérabilité pour laquelle au moins un des chiffres des codes est différent et où la fonction donne l'authentification.

Après l'analyse syntaxique de la section de debug du binaire, la contrainte et la propriété sont traduites et `RobustB` produit l'expression adéquate pour un problème SMT.

Le listing B.2 montre la modélisation de la propriété. La fonction `(_ zero_extend X)` étend une variable de type `bitvector` de X bits avec des zéros. La fonction `(intern_ldrb A M)` retourne l'octet à l'adresse A de la mémoire M. La mention `(_ bvval 32)` permet de créer une variable de type `bitvector` de 32 bit et contenant la constante val. La propriété exprime donc une lecture à l'adresse de la variable `g_authenticated` (108793 ou `0x1A8F9`) dans la mémoire de l'ensemble `sf` (ensemble final) du chemin fauté et la compare avec la valeur `0xAA` (170 en décimal).

```c
#include "interface.h"
#include "types.h"
#include "commons.h"

BOOL g_authenticated;
SBYTE g_ptc;
UBYTE g_countermeasure;
UBYTE g_userPin[PIN_SIZE];
UBYTE g_cardPin[PIN_SIZE];

BOOL verifyPIN_2(void);

int main()
{
    initialize();

constraint:
__attribute__((annotate("\\assert
              ((g_userPin[0] != g_cardPin[0]) ||
               (g_userPin[1] != g_cardPin[1]) ||
               (g_userPin[2] != g_cardPin[2]) ||
               (g_userPin[3] != g_cardPin[3]));")));

    verifyPIN_2();

property:
__attribute__((annotate("\\assert g_authenticated != 0xAA;")));

    if (oracle()) {
      return 1;
    }
    return 0;
}
```

Listing B.1: code C

```
(assert
  (and
    referencepath
    (not
      (=>
        faulty0path
        (=
          ((_ zero_extend 24) (intern_ldrb (_ bv108793 32) m_sf_faulty0))
          (_ bv170 32)
        )
      )
    )
  )
)
```

Listing B.2: Modélisation de la propriété

Le listing B.3 montre la modélisation de la contrainte lorsqu'elle est exprimée au niveau source. Dans la modélisation, la contrainte est appliquée à l'ensemble `s0000` du chemin de référence, mais elle impacte aussi l'ensemble `s0000` du chemin fauté puisque chaque variable dans l'ensemble `s0000` du chemin de référence est contrainte à l'égalité avec son homologue dans le chemin fauté. les variables des deux ensembles sont contraintes à l'égalité. La fonction `(bvadd A B)` additionne les `bitvectors` A et B et retourne le résultat. La fonction `(bvmul A B)` multiplie le `bitvector` A par B et retourne le résultat. Les valeurs `10875` et `108799` correspondent aux adresses de base respectives des tableaux `g_userPin` et `g_cardPin`.

```
(assert // (or A (or B (or C D))) -> A or B or C or D}
  (or
  // A = *(g\_userPin + (0 * sizeof(char))) != *(g\_cardPin + (0 * sizeof(char)))}
    (not
      (=
        ((_ zero_extend 24)
          (intern_ldrb
            (bvadd (_ bv108795 32) (bvmul (_ bv0 32) (_ bv1 32)))
            m_s0000_reference)
        )
        ((_ zero_extend 24)
          (intern_ldrb
            (bvadd (_ bv108799 32) (bvmul (_ bv0 32) (_ bv1 32)))
            m_s0000_reference)
        )
      )
    )
    (or
      // B = *(g\_userPin + (1 * sizeof(char))) != *(g\_cardPin + (1 * sizeof(char)))}
      (not
        (=
          ((_ zero_extend 24)
            (intern_ldrb
              (bvadd (_ bv108795 32) (bvmul (_ bv1 32) (_ bv1 32)))
              m_s0000_reference)
          )
          ((_ zero_extend 24)
            (intern_ldrb
              (bvadd (_ bv108799 32) (bvmul (_ bv1 32) (_ bv1 32)))
              m_s0000_reference)
          )
        )
      )
      (or
        // C = *(g\_userPin + (2 * sizeof(char))) != *(g\_cardPin + (2 * sizeof(char)))}
        (not
          (=
            ((_ zero_extend 24)
              (intern_ldrb
                (bvadd (_ bv108795 32) (bvmul (_ bv2 32) (_ bv1 32)))
                m_s0000_reference)
            )
            ((_ zero_extend 24)
              (intern_ldrb
                (bvadd (_ bv108799 32) (bvmul (_ bv2 32) (_ bv1 32)))
                m_s0000_reference)
            )
          )
        )
        // D = *(g\_userPin + (3 * sizeof(char))) != *(g\_cardPin + (3 * sizeof(char)))}
        (not
          (=
            ((_ zero_extend 24)
              (intern_ldrb
                (bvadd (_ bv108795 32) (bvmul (_ bv3 32) (_ bv1 32)))
                m_s0000_reference)
            )
            ((_ zero_extend 24)
              (intern_ldrb
                (bvadd (_ bv108799 32) (bvmul (_ bv3 32) (_ bv1 32)))
                m_s0000_reference)
            )
          )
        )
      )
    )
  )
)
```

Listing B.3: Modélisation de la contrainte

Bibliographie

[ABF+02] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. Fault attacks on rsa with crt : Concrete results and practical countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 260–275. Springer, 2002. 33

[AGL+03] Mehdi-Laurent Akkar, Louis Goubin, Olivier Ly, et al. Automatic integration of counter-measures against fault injection attacks. *Pre-print found at http ://www. labri. fr/Perso/ly/index. htm*, 2003. 18

[And09] Philippe Andouard. *Outils d'aide à la recherche de vulnérabilités dans l'implantation d'applications embarquées sur carte à puce*. PhD thesis, Bordeaux 1, 2009. 35, 42

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2) :1–7, 2011. 39

[BBC+14] Maël Berthier, Julien Bringer, Hervé Chabanne, Thanh-Ha Le, Lionel Rivière, and Victor Servant. Idea : embedded fault injection simulator on smartcard. In *International Symposium on Engineering Secure Software and Systems*, pages 222–229. Springer, 2014. 27, 36, 42

[BBK+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes : effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 7. ACM, 2010. 37

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices : Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11) :3056–3076, 2012. 15

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer Science & Business Media, 2013. 28

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011. 84

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3) :50, 2018. 31, 95

[BCR16] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 1–6. ACM, 2016. 19, 79, 106

[BDD+18] Stephan Borel, L Duperrex, E Deschaseaux, J Charbonnier, J Cledière, R Wacquez, J Fournier, J-C Souriau, G Simon, and A Merle. A novel structure for backside protection against physical attacks on secure chips or sip. In *2018 IEEE 68th Electronic Components and Technology Conference (ECTC)*, pages 515–520. IEEE, 2018. 17

[BDML97] D. Boneh, R. A De Millo, and R. J Lipton. On the Importance of Checking Cryptographic Protocols for Faults. *Conference on Theory and application of cryptographic techniques (EUROCRYPT)*, pages 37–51, 1997. 1, 6, 31, 33

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2) :370–382, 2006. 14, 17, 56

[BGV11] J. Balasch, B. Gierlichs, and I. Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 105–114. IEEE, 2011. 15, 56

[BHKTL10] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky, and J-F Lalande. Attack model for verification of interval security properties for smart card c codes. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, page 2. ACM, 2010. 32, 42

[BHKTL12] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky, and Jean-Francois Lalande. High level model of control flow attacks for smart card functional security. In *2012 Seventh International Conference on*

Availability, Reliability and Security, pages 224–229. IEEE, 2012. 16, 28, 32, 42

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *International Conference on Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011. 16

[BJC15] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the back-side fault attacks : With a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 99–103. ACM, 2015. 56

[BKL+07] A. Bogdanov, L. R Knudsen, G. Leander, C. Paar, A. Poschmann, M. JB Robshaw, Y. Seurin, and C. Vikkelsoe. Present : An ultra-lightweight block cipher. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007. 38

[BOS03] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new crt-rsa algorithm secure against bellcore attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 311–320, New York, NY, USA, 2003. ACM. 15

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691, 1986. 28

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard : Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010. 84

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In *International Conference on Rewriting Techniques and Applications*, pages 76–87. Springer, 2003. 35

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee : Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. 34

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3) :215–222, 1976. 30

[CMD+18] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory : Instructions corruption on a 32-bit microcontroller. *IACR Cryptology ePrint Archive*, 2018 :1042, 2018. 12, 15, 56

[Cri] Common Criteria. Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 5. 25

[DMB08] L. De Moura and N. Bjørner. Z3 : An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. 84, 98

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories : introduction and applications. *Communications of the ACM*, 54(9) :69–77, 2011. 30

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the aes round counter. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 17–31. Springer, 2013. 6, 12

[DPP+16] L. Dureuil, G. Petiot, M-L. Potet, A. Crohen, and P. De Choudens. FISSC : a Fault Injection and Simulation Secure Collection. In *International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 3–11, 2016. 18, 98

[Dur16] Louis Dureuil. *Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute*. PhD thesis, 2016. 12, 19, 36, 42, 105, 131, 138

[EBRM16] David El-Baze, Jean-Baptiste Rigaud, and Philippe Maurine. A fully-digital em pulse detector. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 439–444. IEEE, 2016. 17

[Fau08] Olivier Faurax. *Évaluation par simulation de la sécurité des circuits face aux attaques par faute*. PhD thesis, 2008. 27, 39, 42

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas : A taint-based approach. In *International Conference on Computer Aided Verification*, pages 294–313. Springer, 2018. 84

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler : An efficient, scalable and thorough preprocessing. In *LPAR*, pages 363–380, 2018. 127

[GD07] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*, pages 519–531. Springer, 2007. 84

[GHEDK16] L. Goubet, K. Heydemann, E. Encrenaz, and R. De Keulenaer. Efficient design and evaluation of countermeasures against fault attacks using formal verification. In *14th International Conference on Smart Card Research and Advanced Applications (CARDIS)*, pages 177–192, 2016. 37, 42, 43

[Gho18] Marjan Ghodrati. *Thwarting Electromagnetic Fault Injection Attack Utilizing Timing Attack Countermeasure*. PhD thesis, Virginia Tech, 2018. 8, 9

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js : A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016. 10

[God11] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 258–269, 2011. 30

[Hum06] T Humphreys. Iso/cei 27001 : 2005–l'état de l'art en gestion de sécurité de l'information. *ISO Management Systems*, 1 :15–18, 2006. 1

[Jaf19] Nisrine Jafri. *Formal fault injection vulnerability detection in binaries : a software process and hardware validation*. PhD thesis, 2019. Thèse de doctorat dirigée par Lanet, Jean-Louis et Legay, Axel Informatique Rennes 1 2019. 38, 42, 91, 94, 95, 132

[JAR+95] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into vhdl models : the mefisto tool. In *Predictably Dependable Computing Systems*, pages 329–346. Springer, 1995. 38, 42

[KBB+18] Dilip SV Kumar, Arthur Beckers, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of laser pulses on atmega328p. In *International Conference on Smart Card Research and Advanced Applications*, pages 156–170. Springer, 2018. 56

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them : An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3) :361–372, June 2014. XI, 9, 10

[KMW17] M. S. Kelly, K. Mayes, and J. F. Walker. Characterising a cpu fault attack model via run-time data analysis. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 79–84, May 2017. 15, 56

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Fault attacks for crt based rsa : New attacks, new results, and new countermeasures. In *IFIP International Workshop on Information Security Theory and Practices*, pages 215–228. Springer, 2007. 138

[Law96] Kevin P Lawton. Bochs : A portable pc emulator for unix/x. *Linux Journal*, 1996(29es) :7, 1996. 39

[LBD+19] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a risc-v processor. *Microprocessors and Microsystems*, 71 :102862, 2019. 136

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking : An overview. In *International conference on runtime verification*, pages 122–135. Springer, 2010. 30

[LH07] Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In *International Verification Workshop (VERIFY)*, volume 259, pages 85–103, 2007. 32, 42

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card c codes. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2014. 20, 32, 101

[MDH+13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 77–88, 2013. 12, 15, 56, 106

[MHER14] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3) :145–156, February 2014. 56

[MMLC11] Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet, and Julien Cartigny. Smartcm a smart card fault injection simulator. In *2011 IEEE International Workshop on Information Forensics and Security*, pages 1–6. IEEE, 2011. 34, 42

[Mor14] Nicolas Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Paris 6, 2014. 18, 79

[NPB15] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9 :53–58, 2015. 84

[PHBC17] J. Proy, K. Heydemann, A. Berzati, and A. Cohen. Compiler-assisted loop hardening against fault attacks. *ACM Transactions on Architecture and Code Optimization*, 14(4) :36, 2017. 20, 27, 33, 36, 37, 42, 107

[PMPD14] M-L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart : A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 213–222, 2014. 34, 42, 79, 91, 93

[PNKI08] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Symplfied : Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 472–481. IEEE, 2008. 35, 94

[PNKI09] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Discovering application-level insider attacks using symbolic execution. In *IFIP International Information Security Conference*, pages 63–75. Springer, 2009. 35, 42, 91, 94

[Pro19] Julien Proy. Sécurisation systématique d'applications embarquées contre les attaques physiques. 2019. 19, 41, 107

[PTL+11] François Poucheret, Karim Tobich, Mathieu Lisarty, L Chusseauz, B Robissonx, and Philippe Maurine. Local and direct em injection of power into cmos integrated circuits. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 100–104. IEEE, 2011. 9

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy current for Magnetic Analysis with Active Sensor. In *Esmart 2002, Nice, France*, 9 2002. 8

[Quy14] Nguyen Anh Quynh. Capstone : Next-gen disassembly framework. *Black Hat USA*, 2014. 78

[RDT13] Cyril Roscian, Jean-Max Dutertre, and Assia Tria. Frontside laser fault injection on cryptosystems-application to the aes'last round. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 119–124. IEEE, 2013. 7

[RG14] Pablo Rauzy and Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on crt-rsa. *Journal of Cryptographic Engineering*, 4(3) :173–185, 2014. 33, 42

[RNR+15] L. Rivière, Z. Najm, P. Rauzy, J-L. Danger, J. Bringer, and L. Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *IEEE International Symposium on Hardware Oriented Security and Trust, (HOST)*, pages 62–67, 2015. 15

[RPL+14] Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne, and Maxime Puys. Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In *International Symposium on Foundations and Practice of Security*, pages 92–111. Springer, 2014. 36

[RSDT13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in sram memory cells. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 89–98. IEEE, 2013. 56

[SA02] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002. 7

[Sch02] Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2, 3) :111–126, 2002. 28

[SHD+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. Fail* : An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255. IEEE, 2015. 39, 42

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000. 29

[SWS+16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok :(state of) the art of war : Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016. 77

[TK10] Elena Trichina and Roman Korkikyan. Multi fault laser attacks on protected crt-rsa. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 75–86. IEEE, 2010. 138

[TM17] N. Timmers and C. Mune. Escalating privileges in linux using voltage fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8, 2017. 6, 12

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW : Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017. 10

[VHdGC20] Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, and Albert Cohen. Secure delivery of program properties through optimizing compilation. In *In Proceedings of the 29th International Conference on Compiler Construction (CC '20)*, 2020. 84

[VKS11a] I. Verbauwhede, D. Karaklajic, and J-M. Schmidt. The fault attack jungle - A classification model to guide you. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 3–8, 2011. 16

[VKS11b] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. The fault attack jungle-a classification model to guide you. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–8. IEEE, 2011. 15

[Vu17] S. T. Vu. Extraction d'informations d'un code binaire pour l'analyse de sa robustesse vis-à-vis de propriétés de sécurité. In *Stage de fin d'étude, UPMC, Mastère de sciences et technologies mention informatique, spécialité : système électronique système informatique*, 2017. 78

[VWWM11] Jasper GJ Van Woudenberg, Marc F Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 91–99. IEEE, 2011. 6, 12

[WMM04] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of k-path tests for c functions. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 290–293. IEEE Computer Society, 2004. 31

[YGS+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software fault resistance is futile : Effective single-glitch attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016. 8

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software : Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2) :111–130, Jun 2018. XI, 13, 14

[ZDT+14] Loic Zussa, Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, Philippe Maurine, Ludovic Guillaume-Sage, Jessy Clediere, and Assia Tria. Efficiency of a glitch detector against electromagnetic fault injection. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014. 9, 17