Étude de vulnérabilité d'un programme au format binaire en présence de fautes précises et nombreuses

Antoine Gicquel

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Antoine GICQUEL

Étude de vulnérabilité d'un programme au format binaire en présence de fautes précises et nombreuses

Thèse présentée et soutenue à Rennes, le 12/12/2024
Unité de recherche : Centre Inria de l'Université de Rennes

Rapporteurs avant soutenance :

Marie-Laure POTET Professeur, Grenoble INP
Vianney LAPÔTRE Maître de conférences, Université Bretagne Sud

Composition du Jury :

Président :	Jean-Max DUTERTRE	Professeur, École des Mines de Saint-Étienne
Examinateurs :	Guillaume BOUFFARD	Ingénieur de recherche, ANSSI
	Marie-Laure POTET	Professeur, Grenoble INP
	Vianney LAPÔTRE	Maître de conférences, Université Bretagne Sud
Directeur de thèse :	Erven ROHOU	Directeur de recherche, Inria
Encadrant de thèse :	Damien HARDY	Maître de conférences, Université de Rennes

Invité(s) :

Co-directrice de thèse : Karine HEYDEMANN Maître de conférences, Thales et Sorbonne Université

Remerciements

Je souhaite exprimer ma gratitude à toutes les personnes qui ont contribué à l'aboutissement de ce doctorat. Ce manuscrit est le fruit des nombreux échanges et collaborations enrichissantes qui ont jalonné ces trois ans. En plus des aspects scientifiques, cette thèse m'a également offert une précieuse expérience humaine, et je suis profondément reconnaissant envers tous ceux qui m'ont accompagné et soutenu.

En premier lieu, je remercie chaleureusement ma directrice de thèse, Karine Heydemann, et mon directeur, Erven Rohou, ainsi que mon encadrant, Damien Hardy. Votre bienveillance et la qualité de vos conseils ont été des éléments moteurs dans ce travail. Votre accompagnement lors de l'écriture de ce document a été plus que précieux.

Je tiens également à remercier Marie-Laure Potet et Vianney Lapôtre pour l'attention portés à l'évaluation de ce document, ainsi que Guillaume Bouffard pour avoir accepté de faire partie de mon jury de thèse. Je suis également profondément reconnaissant envers Jean-Max Dutertre d'avoir présidé cette soutenance et d'avoir été membre de mon comité de suivi. Mes remerciements vont aussi à Emmanuelle Encrenaz d'avoir été aussi membre du comité de suivi.

Je tiens à remercier chaleureusement l'ensemble de mes collègues pour leur soutien tout au long de cette thèse. Je souhaite tout d'abord exprimer ma gratitude envers François et Laurent pour m'avoir ouvert les portes du monde académique. Mes remerciements vont également à toute l'équipe PACAP, auprès de laquelle j'ai eu la chance de réaliser cette thèse. Merci à Camille, Sara, Hugo, Aurore, Nicolas Ba., Isabelle, Nassim, Caroline, Hector, Virginie, Pierre B., Nicolas Be., Pierre M., Thomas, Xabier, Matthieu, Niels et André. Enfin, j'adresse mes remerciements aux membres des autres équipes qui ont également contribué à cette thèse. Un merci tout particulier à Amélie G., Amélie M., Ludovic C., Ronan et Pierre-Yves.

Pour terminer, je souhaite remercier ma famille et mes amis, qui m'ont soutenue et

accompagnée tout au long de ce doctorat. Je remercie de tout mon cœur mes sœurs, Fanny, Julie et Romane, mon beau-frère Vincent-Michaël ainsi que mes parents et mes grand-parents. Je tiens aussi à remercier chaleureusement mes amis, qui ont été à mes côtés durant cette aventure, notamment Jade, Ludovic J., Matthias, Vincent, Coline, Yohann, Fabian, Arthur, Lucie, Maxime, Robin et Killian. Chacun d'entre vous a contribué à rendre cette aventure plus joyeuse. Enfin, je tiens à te remercier Nolwenn pour tes encouragements et pour avoir veillé aussi bien sur moi.

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION

Les systèmes embarqués sont omniprésents dans notre quotidien : qu'il s'agisse d'un téléphone portable, d'une voiture connectée ou d'un terminal de carte bancaire, l'opérabilité de ces appareils et la protection des données sensibles qu'ils manipulent sont critiques. Au cours de la dernière décennie, l'émergence des objets connectés a fait apparaitre de nombreux systèmes dont l'accès physique n'est souvent pas contrôlé. Ces appareils intègrent des logiciels pouvant être complexes comme des systèmes d'exploitation complets, avec des interconnexions vers d'autres systèmes, notamment par le biais de connectivités réseaux. Leur sécurité repose non seulement sur la robustesse théorique des programmes, mais aussi celle de leur implémentation logicielle et celle du matériel.

Un système embarqué est potentiellement constitué de nombreux microprocesseurs, plus d'une centaine pour une voiture moderne par exemple, chacun étant potentiellement vulnérable à des attaques pouvant compromettre la sécurité de l'ensemble de la chaîne. Les travaux séminaux de Boneh et al. [1] ont montré qu'une perturbation physique durant l'exécution d'un programme peut altérer son comportement nominal. Ces perturbations, appelées « fautes », peuvent être introduites par un attaquant de diverses manières : un faisceau lumineux tel qu'un laser peut ioniser une partie du circuit, un champ électromagnétique dirigé peut induire des courants transitoires, ou encore la manipulation du signal d'alimentation ou d'horloge de l'appareil peut perturber son fonctionnement.

Les injections de fautes permettent d'introduire des erreurs dans les calculs ou de modifier les données manipulées par un programme. Ces fautes peuvent être exploitées par des attaquants pour compromettre les propriétés de sécurité d'un système. Avec de telles capacités, même un programme prouvé théoriquement robuste peut devenir vulnérable. En raison de la dangerosité que représentent les attaques par injection de fautes, de nombreuses contre-mesures logicielles ont été proposées pour détecter ou limiter leurs effets. Ces protections reposent principalement sur le principe de la redondance [2] : les vérifications ou calculs sensibles sont dupliqués et les constantes sont encodées de telle sorte qu'il soit difficile de modifier leurs valeurs de façon cohérente. De plus, des variables supplémentaires peuvent être ajoutées au code afin de surveiller le flot de contrôle et en vérifier la validité par rapport au programme d'origine [3].

Par conséquent, il est dorénavant nécessaire d'injecter plusieurs fautes, pouvant impacter potentiellement plusieurs instructions, afin de contourner les contre-mesures mises en place et atteindre les objectifs de l'attaque. Les travaux récents sur les moyens d'injection de fautes démontrent que les injections sont plus précises spatialement et temporellement, et qu'il est possible d'injecter plusieurs fautes à des instants différents durant la même attaque [4] ou de corrompre plusieurs instructions consécutives [5]-[10]. Ces attaques sont puissantes et doivent donc être considérées lors d'une évaluation de la sécurité du logiciel et du matériel.

L'évaluation de la sécurité matérielle d'un appareil repose nécessairement sur de véritables campagnes d'injection de fautes [11]. Cependant, les analyses de programme visant à identifier des chemins d'attaque sont souvent utilisées en amont, comme une étape préliminaire d'évaluation, avant même que le système final ne soit disponible. Déterminer les potentiels chemins d'attaque permet de détecter les faiblesses des contre-mesures mises en place et également de réduire le temps de préparation nécessaire à une campagne d'injection de fautes. Ces analyses peuvent opérer à différents niveaux d'abstraction selon l'effet de faute étudié, allant du niveau du code source à celui de la microarchitecture du microprocesseur ciblé. Plusieurs approches existent pour assister les évaluateurs dans l'identification des chemins d'attaque : 1) la simulation qui permet d'observer les effets des fautes en utilisant des valeurs concrètes [12], [13] ; 2) l'exécution symbolique, où le programme est exécuté avec des entrées et des paramètres symboliques incluant les fautes, plutôt que des valeurs concrètes [14], [15] ; 3) les analyses créant un modèle formel du programme, suivies par l'application de méthodes de vérification de modèles pour prouver la validité des propriétés de sécurité en présence de fautes [16]-[18].

Toutefois, la majorité de ces techniques d'analyse cherchent à vérifier de manière exhaustive les effets des fautes afin de détecter les potentielles vulnérabilités. Ainsi, ces techniques se heurtent au problème d'explosion combinatoire lorsque de trop nombreuses fautes sont considérées ou si le code analysé est trop grand. Un décalage commence à apparaître entre les capacités des plateformes d'injection modernes, qui permettent d'introduire plusieurs fautes durant la même attaque et les techniques d'analyses, qui restent quant à elles majoritairement limitées à un faible nombre de fautes. Il devient nécessaire de trouver de nouvelles approches pour analyser la sécurité du programme et du matériel d'un système en présence de fautes multiples. Cette thèse explore d'abord la faisabilité d'une méthode de détermination de chemins d'attaque pouvant passer à l'échelle avec le nombre de fautes, basée exclusivement sur l'analyse statique. L'utilisation de l'analyse

statique est généralement restreinte à la collecte de données préliminaires aux méthodes présentées précédemment. À notre connaissance, il n'existe aucune approche exclusivement statique dédiée à l'analyse des attaques en fautes.

Concernant la réalisation de campagne d'injections, la mise en œuvre des chemins d'attaque identifiés par l'analyse de programme sur du matériel réel présente plusieurs difficultés. Les chemins d'attaque indiquent les instructions ou les lignes de code à cibler, tandis qu'un banc d'injection de fautes requiert des paramètres physiques, tels que la position de la sonde, la durée et l'intensité la perturbation. Également, les analyses reposent sur des modèles de fautes qui abstraient les effets des fautes ainsi que certains critères de réalisation. De ce fait, une portion des chemins d'attaque identifiés par l'analyse peuvent s'avérer irréalisables sur une cible ou un programme donné. Enfin, les attaques multi-fautes exigent une synchronisation entre les moments d'injection et l'exécution du programme de la carte ciblée. En réponse à ces problématiques, des méthodes doivent être développées afin de faciliter la mise en œuvre des campagnes d'injection dans le but de réduire le temps et l'expertise nécessaire pour l'évaluation matériel d'un système. Dans un second temps, cette thèse étudie une approche capable de prendre en compte les conditions de réalisation des fautes lors de l'analyse du programme, puis de convertir les chemins d'attaque en paramètres temporels. Bien que l'étape de calibration de la plateforme d'injection reste manuelle, une mesure du temps d'exécution du programme en présence de fautes est réalisée afin de déterminer un ensemble de paramètres temporels pour un chemin d'attaque donné. Notre approche utilise ces paramètres temporels pour conduire automatiquement des attaques multi-fautes, comprenant les communications avec la plateforme d'injection et la carte cible ainsi que les injections de fautes et le traitement des résultats.

Contributions

Cette thèse cherche à répondre à la problématique de l'explosion combinatoire en proposant une solution permettant d'évaluer les programmes binaires face à des attaques comportant un grand nombre de fautes précises, ainsi que de réduire le fossé entre l'analyse de programme et la mise en œuvre d'une campagne d'injection. Plus précisément, nous proposons les deux contributions suivantes :

— SAMVA : Une analyse statique au niveau assembleur, conçue pour identifier des chemins d'attaque permettant de détourner le flot de contrôle initial. SAMVA utilise des heuristiques afin de pouvoir passer à l'échelle lorsque l'on considère les attaques

multi-fautes. Conçu pour être modulaire et extensible, SAMVA considère des modèles de fautes basés sur le saut d'instructions et des attaques d'accessibilité. Ces travaux ont mené à une publication d'un article dans le workshop international COSADE en 2023 [19].

— SAMPLAI : Une méthode permettant de faire le lien entre les analyses de programme et la réalisation des attaques multi-fautes. SAMPLAI comprend une extension de SAMVA capable de prendre en compte les conditions spécifiques de réalisation des effets des fautes, une technique de conversion des chemins d'attaque en paramètres temporels pour l'injection des fautes et une technique pour automatiser l'exécution des attaques multi-fautes. Ces travaux ne sont pas encore publiés.

Chaque contribution est accompagnée d'une évaluation qui consiste à attaquer des programmes de vérification de code PIN. Ces programmes sont issus de la suite de benchmarks FISCC [20] et chaque version intègre une sélection de contre-mesures différente. Pour l'évaluation de SAMVA, nous identifions des chemins d'attaque sur l'ensemble des versions qui sont ensuite validés à l'aide d'un simulateur. Pour SAMPLAI, nous utilisons l'analyse pour trouver des chemins d'attaque qui respectent les nouvelles contraintes sur le positionnement des fautes, puis les utilisons pour mener des campagnes d'injection de fautes sur ces programmes sur des cibles réelles. Pour nos expérimentations, nous injectons les fautes en perturbant le signal d'horloge à l'aide la plateforme TRAITOR **traitor** et nous ciblons des cartes STM32F1 embarquant un processeur Cortex-M3. Nos résultats montrent que notre méthode est capable de trouver des paramètres de fautes menant à des attaques réussies, contenant jusqu'à huit fautes qui permettent de compromettre l'exécution de plus de 80 instructions.

Organisation du document

L'organisation de ce document est la suivante. Le chapitre 1 introduit les principes de sécurité informatique et les attaques par injection de fautes. Le chapitre 2 présente l'état de l'art des analyses permettant de mesurer le niveau de vulnérabilité d'un programme en présence de fautes. Nous introduisons dans le chapitre 3 notre méthode d'analyse statique SAMVA, permettant d'identifier dans un programme au format binaire des chemins d'attaque qui requièrent des fautes précises et nombreuses. Dans le chapitre 4 nous présentons SAMPLAI, notre méthode permettant de faire le lien entre l'analyse et la réalisation des attaques multi-fautes.

Chapitre 1

Contexte

La notion de sécurité en informatique est vaste, en raison de la diversité des propriétés à garantir et des nombreux vecteurs d'attaque possibles. Dans la section 1.1, nous commençons par donner la définition de plusieurs propriétés de sécurité couramment étudiées par les évaluations de sécurité des systèmes, suivies d'une introduction aux trois principales familles d'attaques informatiques. Dans la section 1.2, nous expliquons plus précisément le fonctionnement des attaques par injection de fautes, qui sont le type d'attaque étudié dans cette thèse. Enfin, la section 1.3 évoque les évolutions des techniques d'injection de fautes et les raisons pour lesquelles il devient nécessaire d'intégrer les attaques comportant plusieurs fautes dans les outils d'évaluation du niveau de vulnérabilité d'un système.

1.1 Attaque informatique

Cette section commence par définir de ce qu'est une attaque informatique et les éléments visés. Ensuite, nous donnons une vue d'ensemble des vecteurs d'attaque : d'abord nous présentons les attaques basées sur les vulnérabilités logicielles ; par la suite celles qui sont basées sur les vulnérabilités au niveau de la microarchitecture ; enfin nous nous intéressons aux attaques physiques et, en particulier, aux attaques par injection de fautes.

1.1.1 Définition d'une attaque informatique

Un système informatique est un ensemble organisé de ressources matérielles et logicielles conçu pour automatiser diverses tâches telles que la collecte, le stockage, le traitement et la distribution de l'information. De manière générale, le support matériel de ces systèmes comprend un ou plusieurs microprocesseurs chargés d'exécuter les opérations de traitement, coordonner des unités de stockage pour conserver et manipuler les données, ainsi que gérer les périphériques d'interface pour l'acquisition et la communication de l'in-

formation. Ces systèmes sont capables d'exécuter des algorithmes exprimés sous la forme de programmes informatique, aussi appelés logiciels.

Les systèmes informatiques font partie intégrante de nos vies quotidiennes : des systèmes d'information utilisés dans le domaine bancaire ou hospitalier jusqu'aux systèmes électroniques embarqués du milieu automobile ou aéronautique. En raison de la criticité de certaines données pouvant être échangées ou l'impact d'un éventuel dysfonctionnement, la sécurité de tels systèmes est primordiale. D'après le magazine *Cybersecurity Ventures*, le coût global de la cybercriminalité en 2021 est estimé à six milliards de dollars [21]. Cela comprend, entre autres, la destruction de données ainsi que le vol d'argent, de propriétés intellectuelles ou encore de données personnelles et financières. En plus du coût financier, l'atteinte à la sécurité informatique peut aussi avoir un coût humain, si on prend l'exemple d'un dysfonctionnement d'un programme d'autopilotage d'un avion ou d'une voiture.

Les notions fondamentales de sécurité sont apparues dans le cadre des premières études sur la façon d'évaluer la sécurité des systèmes d'information en 1977 réalisées par l'Institut national des normes et de la technologie des États-Unis (NIST) [22]. La sécurité d'un système informatique y est définie comme la protection contre les accidents ou des menaces délibérées, résumée par la garantie du respect de ces trois propriétés clefs :

— **Confidentialité (*Confidentiality*)** : l'accès ou le partage de données n'est possible que pour les utilisateurs ou entités autorisés.

— **Intégrité (*Integrity*)** : les données ou ressources du système sont fiables, précises et cohérentes. Par opposition, un système qui n'est pas intègre signifie qu'un agent non autorisé est capable de modifier, altérer ou supprimer des données.

— **Disponibilité (*Availability*)** : le système informatique est accessible et opérationnel lorsque qu'un utilisateur ou une entité autorisés le demande.

Également appelée la triade « CIA » (pour l'acronyme en anglais), ces trois concepts sont devenus une référence pour l'évaluation de la sécurité des systèmes. Cependant, avec l'évolution constante des technologies, de nouvelles propriétés ont émergé pour enrichir les techniques d'évaluation. L'authentification, par exemple, qui assure l'identité des utilisateurs ou entités accédant aux ressources. De même, la non-répudiation est aussi devenue une propriété intéressante en garantissant qu'un utilisateur ou une entité ne peut pas dénier ses actions réalisées au sein du système. Dorénavant, de nombreuses normes internationales sont en places telles que l'ISO/CEI 27000 en 2005 et l'ISO/IEC 15408 en 1998, appelée plus simplement « Common Criteria » et renouvelée tous les quatre ans [11]. Ces

normes sont promues par des politiques nationales et européennes, notamment en France avec la directive NIS 2 [23]. L'objectif est de formaliser les propriétés de sécurité pour un système informatique et s'accorder sur les garanties qu'elles offrent.

Une attaque informatique est donc la tentative volontaire de violer l'une de ces propriétés de sécurité. Un attaquant cherche à perturber un des éléments d'un système dans le but d'obtenir, par exemple, des données sensibles, briser des propriétés cryptographiques ou simplement prendre le contrôle sur le système entier. Pour réaliser de telles attaques, un adversaire peut choisir plusieurs vecteurs d'attaque que nous détaillons dans la suite de cette section.

1.1.2 Attaque basée sur une vulnérabilité logicielle

De la conception jusqu'à la mise en production d'un logiciel, il existe une multitude d'étapes où des failles de sécurité peuvent apparaître. Un programme informatique est d'abord écrit par des humains sous forme de code source, pouvant laisser des bogues ou des erreurs algorithmiques. Puis ce code source est compilé en code binaire compréhensible par la machine qui l'exécute, pouvant également intégrer des failles qui ne seront visibles qu'à l'exécution [24]. Le programme peut également nécessiter divers types de configurations ou de connexions réseau, qui, si elles sont mal réalisées, peuvent créer des failles de sécurité. Ainsi, une vulnérabilité logicielle signifie la possibilité de compromettre les propriétés de sécurité par l'exécution du programme cible et ses interactions avec son environnement. Pour illustrer ce type d'attaque, nous présentons deux exploitations ayant eu des impacts importants.

La vulnérabilité « Heartbleed » en 2014 a affecté la bibliothèque cryptographique OpenSSL, qui est largement utilisée pour le chiffrement des communications sur Internet avec le protocole SSL/TLS. L'origine de la faille est une erreur de programmation, où la taille d'une lecture de la mémoire n'est pas vérifiée. Par conséquence, un attaquant pouvait envoyer au serveur une requête spécialement conçue, et celui-ci répondait en exposant des données potentiellement sensibles, tels que des clefs privées, des noms d'utilisateurs ou des mots de passe [25].

Un autre exemple marquant sont les techniques « ROP » pour Return-Oriented Programming. Les programmes sont compilés de sorte que les fonctions se terminent par une instruction permettant de quitter la procédure (*ret* en assembleur x86) et poursuivre l'exécution à l'adresse stockée dans la pile. Les attaques ROP reposent sur un défaut dans la conception du programme, notamment sur les accès en lecture et écriture dans

la mémoire, pour obtenir le contrôle de la pile d'exécution (aussi appelée *call stack*). En réécrivant certaines adresses de retour en mémoire, l'attaquant peut manipuler le flot de contrôle et exécuter une suite de courtes séquences d'instructions du programme original, finement sélectionnées, afin de réaliser des exploits [26].

1.1.3 Attaque basée sur une vulnérabilité microarchitecturale

La microarchitecture d'un processeur désigne sa description au niveau logique qui détaille la manière dont le jeu d'instructions est implémentée. Le jeu d'instructions noté « ISA » pour *Instruction Set Architecture* représente l'ensemble des instructions exécutables par le processeur et sont encodées par un code opération abrégé *opcode*. Les instructions permettent d'effectuer des opérations arithmétiques, des lectures et écritures en mémoire, des branchements, ainsi que des interactions avec le système. La description de la microarchitecture comporte la chaîne de traitement (*pipeline* en anglais) et sa profondeur. Cette chaîne décrit comment les instructions sont exécutées, les différentes mémoires cache ou certains traitements préliminaires appliqués aux instructions.

Afin d'augmenter les performances des processeurs ou diminuer leur consommation électrique, de nombreuses optimisations sont implémentées dans les processeurs modernes. Les plus performants sont particulièrement complexes et il est aujourd'hui difficile de les comprendre dans leur entièreté. Une modification locale peut avoir des effets de bord difficiles à appréhender. Pour cette raison, la microarchitecture peut aussi devenir un vecteur d'attaque puissant mettant à mal la sécurité du système.

La vulnérabilité nommée Spectre [27] illustre bien le risque que représentent les failles de sécurité au niveau microarchitectural. Celle-ci exploite les mécanismes d'exécution spéculative, notamment quand le processeur essaie de prédire les futures instructions pour les exécuter en avance afin d'améliorer les performances. En exploitant les écarts temporels qui se produisent pendant l'exécution spéculative, un attaquant peut inférer le contenu d'emplacement mémoire contenant des données sensibles auxquels il n'a pas accès directement. Cette faille a été très médiatisée, car elle touche la plupart des processeurs d'application utilisés dans l'industrie. Ce type de faille est difficile à corriger de manière logicielle et se passer de ce genre d'optimisation impacterait trop significativement les performances.

1.1.4 Attaque physique

Les attaques physiques exploitent des caractéristiques matérielles pour violer les propriétés de sécurité d'un système. À cet égard, une implémentation correcte d'un protocole sûr n'est plus nécessairement sécurisée. Ces attaques requirent un accès physique au dispositif attaqué et concernent dans la majorité des cas des appareils embarqués comme une carte bancaire, un téléphone portable ou des appareils composant l'internet des objets. Nous présentons les deux catégories d'attaques physiques : les attaques passives où l'on observe et exploite des grandeurs physiques mesurables pendant l'exécution d'un programme sur la cible tandis que les attaques actives consistent à perturber physiquement l'environnement proche des composants pour y introduire des erreurs pendant son exécution.

Attaque physique passive

Les attaques par canaux auxiliaires sont une technique d'attaque passive, basée sur l'observation de grandeurs physiques mesurables comme le temps, la consommation électrique, les émanations électromagnétiques ou la température durant le fonctionnement du système cible. Ces grandeurs mesurables varient en fonction des instructions exécutées et des données manipulées.

La recherche sur les attaques passives a émergé avec le projet *TEMPEST* lancé par la NSA (agence nationale de sécurité américaine) pendant la guerre froide. Les dispositifs analogiques comme des écrans ou des antennes pouvaient être ciblés dans un but d'espionnage. Depuis le papier séminal de Kocher et al. 1999 [28], les attaques par canaux auxiliaires modernes concernent aussi les systèmes numériques. Du fait que les valeurs binaires sont représentées physiquement par des tensions hautes et basses, un 1 nécessite davantage d'énergie qu'un 0 lors d'une opération sur un bit ou inversement selon l'encodage. Les auteurs proposent deux méthodes d'analyse de la consommation suffisamment précises pour casser des implémentations cryptographiques. La première est la *Simple Power Analysis* (analyse de consommation simple ou *SPA*) qui exploite les variations de consommation entre des chemins d'exécutions. Ces chemins d'exécution dépendent du secret et l'analyse des variations permettent de retrouver la valeur du secret. La deuxième méthode est la *Differential Power Analysis* (analyse de consommation différentielle ou *DPA*) qui exploite des différences de consommation pour des valeurs différentes. Pour ce type d'attaque, plusieurs traces sont nécessaires et l'attaque vise un calcul intermédiaire

dépendent du secret et d'une valeur maitrisée par l'attaquant.

Dans le même temps, les techniques *TEMPEST* se sont améliorées. Les travaux de Kuhn [29] ont démontré que les émanations électromagnétiques des connectiques d'un écran peuvent être automatiquement mesurées et analysées à l'aide d'un ordinateur afin de lire son contenu. Cette thèse s'intéresse aux attaques par injection de fautes et ne traite pas de ce type d'attaque.

Attaque physique active

Les attaques actives consistent à introduire volontairement un stress physique dans les composants d'un système pendant son exécution pour y induire des fautes, c'est-à-dire des changements de comportement tels que des erreurs de calcul ou des modifications des données manipulées. En injectant des fautes, un attaquant peut altérer l'exécution d'une portion critique du programme, compromettant ainsi la sécurité du système. Les composants matériels peuvent être perturbés par divers moyens, par exemple, en utilisant une émission laser ou une impulsion électromagnétique.

L'attaque *BellCoRe* (pour Bell Communications Research), introduite en 1997 par Boneh et al. [1], est la première application d'attaque par injection de fautes. Cette dernière démontre comment des erreurs induites lors des calculs cryptographiques d'une implémentation de l'algorithme de chiffrement RSA-CRT permettent d'extraire la clef privée. Cet article fondateur a permis d'exposer la criticité de l'impact potentiel des fautes lors de la conception de système ainsi qu'ouvrir la voie à de nombreux travaux de cryptanalyse en présence de faute. La même année, Biham et al. [30] formalise la méthode d'analyse différentielle de faute (ou DFA pour *Differential Fault Analysis* en anglais), montrant qu'il est possible d'exploiter le fait qu'une faute spécifique peut impacter les états observables du système en les comparant avec ceux obtenus après une exécution nominale. Les auteurs illustrent l'efficacité de la technique en attaquant une implémentation du chiffrement DES, puis le chiffrement AES sera prouvé vulnérable au DFA par Piret et al. [31]. En complément, la méthode d'analyse de faute sans effet (ou *Safe Error Analysis* en anglais) par Yun et al. [32] repose sur le fait qu'une faute ne mène pas toujours à une sortie fautée et qu'on peut en déduire des informations sur la valeur d'un secret, telle que la clef privée du chiffrement RSA.

Depuis ces travaux initiaux ciblant principalement les implémentations cryptographiques, le domaine de l'injection de fautes s'est considérablement sophistiqué. Les moyens d'injection sont désormais plus précis et permettent d'injecter plusieurs fautes. De plus,

la communauté a acquis une meilleure compréhension des effets des fautes, ce qui a élargi leur exploitation à davantage de cibles. Même les contre-mesures logicielles et autres mesures de sécurité peuvent aussi être les cibles des attaques en fautes, à condition de savoir où et quand les injecter. À ce titre, cette thèse s'intéresse aux attaques par injection de fautes, et plus précisément à l'analyse des programmes au format binaire afin de déterminer les sections à perturber pour réaliser une attaque. Cela nécessite de comprendre précisément les attaques possibles et leurs effets ainsi que les protections existantes. C'est l'objet de la section suivante.

1.2 Attaque par injection de fautes

Une attaque par injection de fautes consiste à perturber intentionnellement l'environnement physique d'un système lors de l'exécution d'un programme cible dans le but de compromettre sa sécurité. Ce genre d'attaque s'appuie sur le phénomène de *propagation des fautes*. Les perturbations au niveau physique ont un impact au niveau logiciel. Cette section propose tout d'abord un aperçu des principales techniques d'injection de fautes. Les fautes induites se manifestent ensuite à différents niveaux d'abstraction au cours de leur propagation. Pour chacun de ces niveaux, nous expliquons les effets possibles des fautes. Ensuite, les aspects pratiques de la réalisation des injections de fautes ainsi que la caractérisation des effets de fautes sont abordés. Quelques exemples d'exploitation sont présentés. Enfin, nous décrivons les contre-mesures matérielles et logicielles qui permettent de rendre un système plus résistant aux fautes.

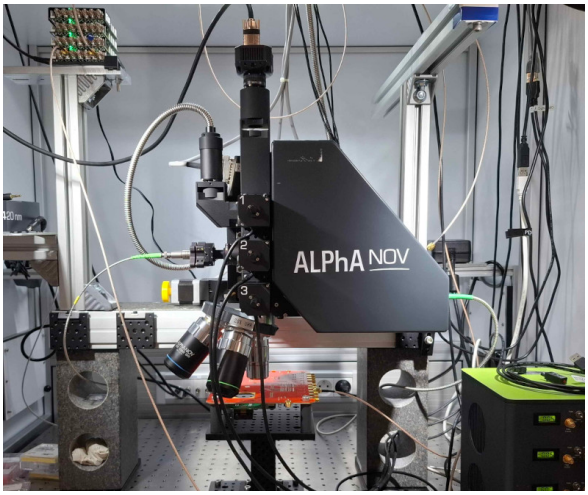
1.2.1 Moyens d'injection de fautes

Les appareils électroniques doivent fonctionner dans un environnement physique susceptible de varier. Lors de leur conception, les constructeurs définissent donc un ensemble de conditions nominales sur l'environnement de fonctionnement. Ces critères sont exprimés sous la forme de plage de valeurs acceptables, concernant entre autres la température, l'humidité, la tension électrique, la fréquence d'échantillonnage. L'objectif d'un moyen d'injection de fautes est d'outrepasser volontairement les zones nominales de fonctionnement des composants pendant une courte durée afin de provoquer des erreurs durant l'exécution. Néanmoins, il est nécessaire de veiller à ce que ces perturbations ne soient pas trop extrêmes pour ne pas engendrer la destruction d'une partie de l'appareil ciblé.

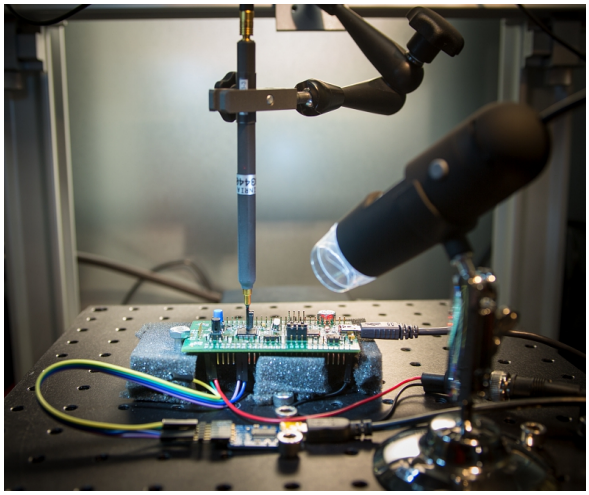
Il existe trois types de fautes : une faute est qualifiée de *permanente* si son effet est irréversible, de *transitoire* si son effet disparait peu après la fin de la perturbation, ou encore de *semi-permanente* si son effet disparait seulement après un redémarrage de l'appareil.

Les différents moyens d'injection de fautes se différencient par la manière dont ils perturbent une cible et par leur précision. Cette précision se décompose en plusieurs aspects : la précision temporelle, qui correspond à la capacité de cibler avec exactitude l'instant de l'injection durant l'exécution et de contrôler sa durée ; et la précision spatiale, qui concerne la capacité à perturber un périmètre spatial choisi. De plus, on évalue également la capacité d'un dispositif d'injection à introduire une ou plusieurs fautes successives, ainsi que le possible délai nécessaire entre deux injections.

Enfin, l'invasivité de la technique d'injection exprime le niveau de modification matérielle à apporter à la cible pour réaliser l'attaque. Une méthode invasive nécessite de retirer une portion de la couche de passivation (substrat de protection) de la puce ciblée pour rendre accessible la matrice de silicium, par opposition, une méthode non invasive ne nécessitant aucune modification. L'intermédiaire entre ces deux techniques est la méthode semi-invasive qui nécessite de décapsuler la puce sans devoir exposer le silicium pour réaliser une attaque.



(a) Plateforme d'injection laser du centre IETR de l'Université de Rennes. ©IETR / L. Claudepierre



(b) Plateforme d'injection EM du Centre Inria de l'Université de Rennes. ©Inria / C. Morel

FIGURE 1.1 – Exemples de bancs d'injection utilisés en laboratoire.

Faisceau optique

Les composants d'un circuit intégré et notamment les transistors sont sensibles aux radiations électromagnétiques. En conséquence, lors d'une exposition à une forte source de rayonnements de lumière comme un laser, les radiations électromagnétiques induites peuvent faire perdre un électron au matériel ciblé. Cela crée un *trou d'électron* et induit, de manière très locale, un courant électrique dans le circuit pouvant provoquer une faute [7], [9], [33]. Afin d'atteindre les circuits, il faut retirer le boitier de la puce (le *package*), rendant cette technique semi-invasive [34].

Un banc d'injection laser est présenté dans la figure 1.1a. Pour cibler une zone précise de la puce, l'attaquant contrôle la position spatiale de la source optique. Le laser est sélectionné afin que le diamètre du faisceau irradie seulement la zone voulue et que sa longueur d'onde permette de traverser efficacement les matériaux intermédiaires. Il est aussi nécessaire de gérer l'intensité de l'injection en paramétrant la puissance de la lumière émise et la durée de l'exposition. Si l'injection est trop intense, la lumière peut brûler le composant qui est alors altéré de manière définitive. Le délai d'activation du laser et la précision de la durée sont typiquement de l'ordre de la nanoseconde [7]. La technique d'injection de fautes par émission laser offre une excellente précision spatiale et temporelle, faisant d'elle une des techniques les plus utilisées. La contrepartie à cette précision est le prix de l'équipement, allant de plusieurs dizaines de milliers d'euros à plusieurs centaines de milliers d'euros.

Les bancs d'injection les plus modernes permettent d'injecter plusieurs fautes pendant la même attaque. D'une part, il existe des techniques *multi-spots* où plusieurs sources d'émission laser visent des positions différentes sur la carte attaquée [4], [35], [36]. D'autre part, les sources d'émission laser peuvent injecter plusieurs fautes sur la même position avec un temps de rechargement entre deux injections d'une durée de plusieurs dizaines de millisecondes [35].

Impulsion électromagnétique

Ce moyen d'injection repose sur l'influence d'un champ magnétique sur un circuit intégré. À l'aide d'un circuit inducteur constitué d'une bobine de fil conducteur, un champ électromagnétique est généré lorsqu'il est traversé par un courant électrique. Des courants électriques, dits *de Foucault*, sont induits par ce champ électromagnétique et vont alimenter les composants, bus et circuits du processeur. Des impulsions électromagnétiques

suffisamment brèves et puissantes peuvent causer des erreurs dans les caches d'instructions [5], [37] et les caches de données [37]. Ces fautes s'observent au niveau logiciel par des sauts d'instructions [5], [8], [38], des substitutions d'opérande ou des corruptions de registre [39]. Bien qu'il n'existe pas encore de consensus précis sur l'explication physique de ces fautes, le courant électrique induit semble causer des violations de contraintes temporelles ou des défauts d'échantillonnage dans les bascules pendant leur phase de commutation [40]. La figure 1.1b montre un banc d'injection électromagnétique.

Un attaquant utilisant un tel banc devra ajuster la taille de la bobine en fonction de la zone ciblée et la positionner au-dessus du circuit intégré. La position inclut aussi la hauteur et l'inclinaison de la source d'émission. Il est possible de précisément contrôler l'intensité de l'injection de fautes en ajustant la tension et le courant électrique alimentant l'inducteur ainsi que la durée de l'impulsion électromagnétique [8]. Cette technique fournit une précision temporelle de l'ordre de la nanoseconde. Concernant la précision spatiale, la bobine irradie une surface plus grande que le laser, celle-ci est au niveau du millimètre et donc légèrement moins précise que les méthodes par injection laser. En revanche, l'injection par impulsion électromagnétique a l'avantage de ne pas être une technique invasive et donc ne nécessite pas de décapsuler la puce ciblée. De plus, l'équipement est un peu moins onéreux qu'un banc laser, mais reste relativement cher comparé aux méthodes par perturbation d'horloge ou de tension.

Perturbation de l'horloge

Un circuit numérique utilise un signal d'horloge pour rythmer son fonctionnement à une fréquence définie. À chacun des cycles d'horloge générés par le composant oscillateur, souvent un *quartz*, les actions du circuit sont réalisées de manière synchrone. Ces actions peuvent intégrer des lectures ou écritures dans la mémoire ou encore l'exécution d'instructions, selon la microarchitecture du processeur considéré. Néanmoins, si un attaquant peut contrôler la fréquence ou la forme du signal d'horloge, alors il peut perturber le fonctionnement du circuit et y provoquer des fautes.

Un tel banc d'injection consiste à remplacer la source du signal d'horloge par un générateur de signaux contrôlables par l'attaquant. Dans un premier temps, le générateur imite le signal d'origine de forme carrée et avec la fréquence de base. Ensuite, l'attaquant peut modifier le signal d'horloge au moment opportun sur une durée définie afin d'injecter des fautes. Il est possible de modifier la fréquence de l'horloge [41]-[43] ainsi que l'amplitude du signal [10].

La précision temporelle de ce type d'attaque est fine, car on peut contrôler la forme de l'horloge pour chacun des cycles. En revanche, la précision spatiale est moins élevée que les autres moyens d'injection puisque les effets des fautes possibles dépendent uniquement du routage des composants vis-à-vis de la source du signal d'horloge [44]. Cette technique est cependant attractive, car l'équipement pour réaliser ce genre d'attaque est peu onéreux, seulement quelques centaines d'euros et permet de facilement réaliser des attaques multi-fautes [10]. L'invasivité de cette méthode dépend de l'accessibilité du quartz d'origine et peut parfois nécessiter des modifications du circuit ciblé pour contourner des mécanismes d'asservissement ou de régulation du signal d'horloge.

Perturbation de l'alimentation

Le fonctionnement des composants d'un circuit numérique requiert une alimentation stable. À l'aide d'une alimentation externe, un attaquant peut perturber la tension électrique d'entrée d'une puce afin de causer des erreurs de fonctionnement lors de l'exécution. De manière générale, les fautes sont occasionnées lors d'une chute de la tension d'alimentation pour une durée comprise entre la nanoseconde et la microseconde. La diminution de la tension a pour conséquence de réduire la vitesse de propagation des signaux dans le circuit et provoque des violations de contraintes temporelles [45].

Les techniques récentes permettent plus de contrôle sur la forme du signal d'alimentation, notamment les fronts montants et descendants de la tension, afin d'injecter des fautes avec davantage d'effets [46]. De façon similaire aux techniques par perturbation d'horloge, cette technique ne permet pas de contrôler spatialement des fautes qui dépendent uniquement du routage des composants. Elle est aussi peu onéreuse, de l'ordre de quelques centaines d'euros et permet aussi des attaques multi-fautes. Enfin, ce type d'injection de fautes n'est pas invasif, néanmoins le fait de retirer les condensateurs externes peut améliorer l'efficacité des attaques [47].

Injection logicielle

Une exécution malicieuse de code peut également engendrer du stress sur le matériel qui l'exécute et introduire des fautes. L'attaque *RowHammer* a démontré qu'en réalisant de nombreuses écritures dans la même ligne de la mémoire vive dans un temps réduit, les cellules environnantes peuvent être perturbées électriquement et changer de valeur (*bit-flip*) [48].

Les mécanismes de gestion de l'énergie des processeurs peuvent aussi être détournés. Afin d'économiser de l'énergie et prolonger la durée de vie de la batterie, il est possible de contrôler logiciellement la fréquence et la tension du matériel. Le *DVFS* (pour *Dynamic Voltage and Frequency Scaling*) peut être exploité pour injecter des fautes, en manipulant soit la fréquence du processeur comme pour l'attaque *CLKSCREW* [49], soit sa tension dans le cas de l'attaque *VoltJockey* [50].

Les moyens d'injection de fautes logiciels sont plus limités que les techniques utilisant un matériel d'injection externe, notamment en ce qui concerne les conditions de réalisation des fautes et leur précision, mais ne nécessitent pas d'accès physique ce qui peut les rendre particulièrement dangereux.

1.2.2 Effets des fautes

Les attaques par injection de fautes se basent sur une perturbation matérielle des circuits intégrés durant leur exécution, en utilisant des moyens d'injection comme ceux cités précédemment, afin de causer un effet au niveau logiciel. Des fautes permanentes peuvent également être injectées lorsque l'appareil est éteint avec des techniques d'altération de données [51]. Au travers d'un mécanisme de *propagation des fautes* comme décrit par Yuce et al. [52], les fautes se manifestent à différents niveaux d'abstraction que nous décrivons dans cette section. La figure 1.2 illustre les étapes de la propagation d'une faute détaillées ci-après.

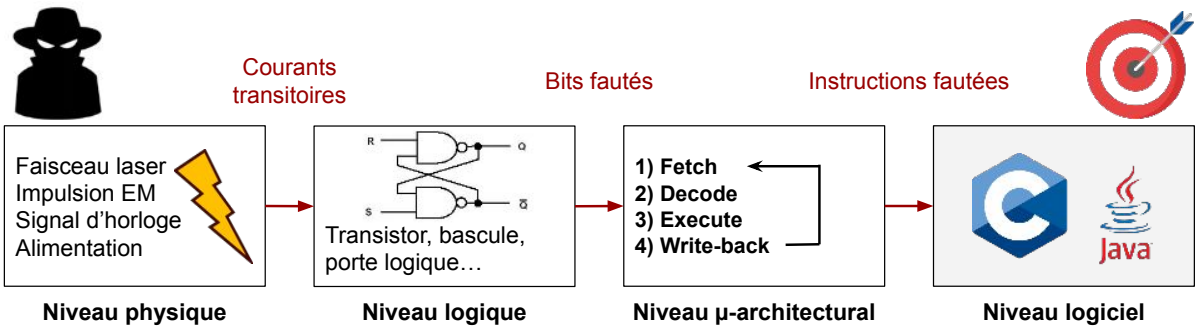


FIGURE 1.2 – Représentation des différents niveaux d'abstraction intervenant lors de la propagation d'une faute.

Manifestation au niveau du circuit

Le fonctionnement d'un processeur est basé sur de la logique séquentielle, dans laquelle les résultats des opérations dépendent des données traitées et des résultats précédents. Cela se traduit par l'usage de composants permettant le stockage de l'information comme des bascules. Les opérations sur les données se font par des vecteurs de bits et se basent sur des combinaisons de portes logiques. L'ensemble de ces composants est constitué de transistors, permettant de contrôler les signaux électriques et leurs formes, mais sont particulièrement sensibles aux perturbations physiques extérieures. La corruption des signaux électriques a un impact au niveau logique sous la forme d'erreurs lors des calculs ou des valeurs erronées.

En fonction de la localisation et l'intensité de la perturbation, l'effet de la faute au niveau du circuit se caractérise par l'altération de bits avec différentes granularités. Par ordre de précision décroissante, il est possible de manipuler la valeur d'un bit, de plusieurs bits distincts, voire d'un octet ou d'un mot. Lorsque plusieurs bits sont manipulés, l'attaquant peut ou non avoir le contrôle sur la valeur eronnée, dans ce dernier cas, la valeur finale peut être aléatoire. Les effets des fautes sur les bits sont le *bit-set* [9], le *bit-reset* [53] et le *bit-flip* pour respectivement la mise à un de la valeur d'un bit, sa mise à zéro et enfin l'inversion de sa valeur.

Propagation de la faute au niveau microarchitectural

Un processeur pipeliné exécute les instructions en plusieurs étapes selon sa microarchitecture, qui définit le nombre et le type des étapes de sa chaîne de traitement. Bien que pouvant varier, l'exécution d'une instruction suit ce schéma général : (i) le processeur charge l'instruction encodée en binaire depuis la mémoire Flash ou la mémoire RAM (étage de *fetch*) (ii) puis détermine les actions à réaliser selon l'opcode de l'instruction chargée et calcule l'adresse de l'instruction suivante (étage de *decode*) (iii) exécute l'instruction courante (étage de *execute*) (iv) enfin, met à jour l'état du processeur avec le résultat dans les registres généraux ou la mémoire (étage de *write-back*).

Comme chaque instruction utilise un sous-ensemble spécifique de la microarchitecture en fonction de son type (par exemple une opération arithmétique ou un chargement de données), l'effet de la faute au niveau logiciel dépend de l'état du processeur. L'altération d'un ou plusieurs bits peut venir changer radicalement son comportement. Par exemple, il est possible de corrompre l'instruction chargée en attaquant l'étage de *fetch*, ou encore

corrompre les données traitées durant les étages *execute* ou *write-back*.

De nombreux travaux ont étudié les effets des fautes sur la microarchitecture [44], [53]-[56]. À l'issue d'une phase de caractérisation, un *modèle de faute* décrit à un niveau donné les effets des fautes sur le traitement de l'instruction ou l'exécution du code source ou encore le fonctionnement d'un algorithme. Ces modèles peuvent ensuite être analysés ou simulés afin de mieux comprendre les conséquences au niveau logiciel et les exploitations envisageables [55].

Observation de la faute au niveau logiciel

De nombreux modèles de fautes au niveau *ISA* décrivent les effets observables des fautes, qui se manifestent sous la forme de modifications des instructions ou des données manipulées.

La corruption d'instructions peut affecter une seule instruction [5], une séquence de plusieurs instructions [7], [8], [34], [57], voire plusieurs séquences de tailles variables dans le cas d'une injection multi-faute [10]. Les effets potentiels de ces corruptions incluent la modification de l'opcode [5], le saut d'instruction (*instruction skip*), qui consiste à ne pas exécuter une instruction [5], ou encore le rejeu d'instruction (*skip-repeat*), qui consiste à réexécuter une instruction à la place d'une ou plusieurs instructions spécifiques qui ne seront alors pas exécutées [10], [43]. La modification d'un opcode peut également entraîner un saut d'instruction lorsque le nouvel opcode désigne une instruction sans effet dans le contexte du programme en cours. Si le jeu d'instructions comporte des instructions à taille variable, une faute peut provoquer un désalignement temporaire de l'exécution, entraînant un décodage incorrect des instructions. De ce fait, l'encodage de deux instructions de petite taille peut être interprété comme une seule et nouvelle instruction, inexistante dans le programme initial [56].

Les effets des altération de bit mentionnées précédemment, telles que le *bit-set*, le *bit-reset* et le *bit-flip*, se manifestent au niveau logiciel par des modifications des valeurs des registres et des variables [38] ou code. Ces modifications peuvent affecter seulement une portion d'une variable avec une granularité allant du bit à l'octet, en fonction de la précision de l'attaquant et la taille de la variable ciblée.

Les effets des fautes peuvent aussi être modélisés au niveau du code source. Les fautes peuvent notamment altérer les structures de contrôle comme des inversions des branchements conditionnels [14], contrôler le nombre d'itérations d'une boucle [58] ou encore sauter l'exécution de ligne de code [59]. Bien qu'insuffisant pour réaliser une analyse de

sécurité du système, ce niveau d'abstraction permet de raisonner sur les effets possibles au niveau source et déployer des protections.

En conclusion, les effets des fautes au niveau logiciel permettent de corrompre le flot de contrôle ou le flot de données d'un programme. Ces perturbations de l'exécution peuvent être exploitées pour attaquer un système.

1.2.3 Réalisation pratique de l'injection de fautes

Réaliser une attaque par injection de fautes requiert de paramétrer le banc d'injection dans le but de provoquer une ou plusieurs perturbations exploitables. Néanmoins, cette étape préliminaire peut être chronophage et nécessiter une expertise technique considérable. Nous introduisons cette première étape de *calibration* du moyen d'injection permettant d'optimiser la recherche de paramètres. Par la suite, nous présentons l'intérêt d'une campagne de caractérisation qui étudie le lien entre les paramètres du banc, les conditions de l'attaque et les effets des fautes. La caractérisation permet de mieux comprendre les effets des fautes et d'améliorer la reproductibilité des attaques. Elle peut également servir ultérieurement à des méthodes d'analyse de la sécurité du système ou à la conception de protections.

Paramétrage du moyen d'injection

Un banc d'injection de fautes comme ceux basés sur l'usage de laser ou d'impulsion électromagnétique peut nécessiter une longue étape de calibration. Une des techniques souvent employées pour déterminer des paramètres spatiaux (sur les axes x et y) est de balayer une zone au-dessus d'une puce électronique avec un pas de précision variable, puis d'observer si le point d'injection occasionne une faute au niveau logiciel ou un plantage. Cependant, en plus d'être chronophage, cette méthode peut devenir impossible s'il faut prendre en compte d'autres paramètres comme l'intensité des fautes, l'éventuelle forme de la courbe de l'impulsion et les effets combinatoires en cas de fautes multiples, voire plusieurs points d'injections. De ce fait, des travaux ont été menés dans le bus d'optimiser la calibration des bancs.

L'espace des paramètres pour un équipement donné est généralement exploré à l'aide de ces trois méthodes : 1) recherche par grille (*Grid Search*), une technique semi-exhaustive sur une plage de valeurs prédéterminées qui est ensuite progressivement affinée ; 2) recherche aléatoire (*Random Search*), qui peut prendre du temps à trouver des paramètres

satisfaisants, mais permet de mieux passer à l'échelle lorsque l'espace des paramètres est plus grand ; 3) les techniques utilisant des optimisations particulières telles que les algorithmes métaheuristiques ou des hyperparamètres [60], qui restent efficaces même pour des espaces avec un grand nombre de dimensions.

La calibration temporelle entre l'équipement d'injection et le matériel ciblé repose sur un signal de synchronisation. Lors d'une attaque réelle, ce signal est généralement issu d'une analyse préalable de la consommation électrique ou des émanations électromagnétiques. En contexte d'évaluation, il est courant d'implémenter un signal factice sur l'une des broches de la carte ciblée, s'activant juste avant l'exécution du code que l'on souhaite attaquer.

Caractérisation des effets des fautes

La caractérisation est le processus d'identification des effets possibles et observables des fautes pour des programmes spécifiques, permettant d'inférer des modèles de fautes. La figure 1.3 représente le procédé général d'une caractérisation qui consiste à comparer des exécutions en présence de fautes avec une exécution nominale sans faute (*golden run*). Cette comparaison se fait sur des valeurs connues et contrôlables du système comme les valeurs des registres, des variables en mémoire ou encore des signaux accessibles de la microarchitecture. En connaissant l'état initial du programme et les possibles états finaux différents, des modèles de fautes peuvent être inférés, par exemple, une altération de donnée ou un saut d'instruction. Les états finaux dépendent des paramètres du moyen d'injection de fautes, et donc la caractérisation permet de comprendre l'influence de la position de l'équipement ou l'intensité de la perturbation sur les effets causés. Pour permettre une bonne identification des effets des fautes, des programmes de test doivent être conçus afin d'isoler les potentiels effets des fautes pour permettre l'inférence des modèles. Cette analyse est souvent manuelle et donc chronophage. De plus, les résultats sont très sensibles aux codes de test utilisés.

1.2.4 Exploitation des fautes

Pour un attaquant, l'exploitation des fautes a pour but final de compromettre la sécurité d'un appareil. Historiquement, les attaques par injection de fautes ont surtout concerné les protocoles cryptographiques comme celle sur une implémentation d'algorithme de chiffrement RSA par Boneh et al. [1]. Par la suite, des implémentations de

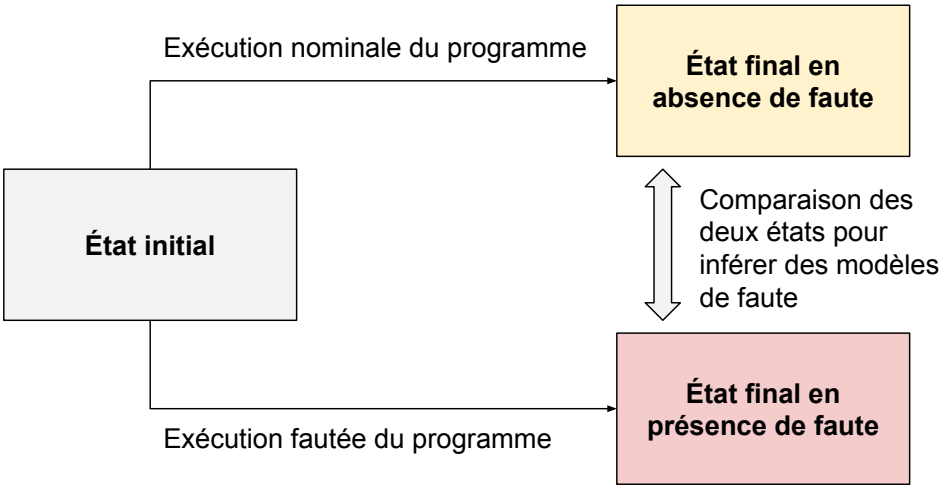


FIGURE 1.3 – Principe général de la caractérisation des effets des fautes.

démarrage sécurisé (mieux connu sous le terme de *secure boot* en anglais) et des vérifications d'authentification ont été ciblées. Désormais avec l'évolution des techniques d'injection et d'exploitation des fautes, même des équipements complexes utilisés dans l'industrie peuvent être impactés [47], [61]. Pour illustrer la menace que représente ce type d'attaque, nous présentons trois exemples d'exploitation issus du « monde réel ».

Un débordement de tampon (ou *buffer overflow* en anglais, abrégé BOF) est l'exploitation d'une vulnérabilité logicielle permettant d'écrire des données au-delà d'une zone mémoire dédiée. Ces données sont écrites en mémoire, dans une variable ou un tableau dans des langages haut niveau, jusqu'à dépasser sa capacité, entrainant une écriture dans la zone mémoire adjacente. Cette vulnérabilité permet de monter des attaques plus complexes en écrivant du code malicieux en mémoire et prendre le contrôle du programme [62]. Les attaques BOF sont bien documentées, et de nombreuses contre-mesures ont été développées pour vérifier la cohérence entre la taille des entrées et celle des tampons mémoire. Toutefois, les travaux de Nashimoto et al. [57] ont démontré expérimentalement qu'il était possible de créer une vulnérabilité permettant de réaliser un BOF à l'aide de plusieurs injections de fautes. Les auteurs ont montré qu'il était possible de contourner l'exécution d'une courte séquence d'instructions correspondant à la vérification de la taille maximale de remplissage du tampon et donc de réaliser des écritures malicieuses en mémoire. Cela montre comment les effets des fautes peuvent détourner des protections ou créer des vulnérabilités, permettant de retomber dans le scénario d'attaque logicielle. L'exploitation

de cette association entre attaque matérielle et logicielle augmente considérablement la surface d'attaque à considérer [6].

Un autre exemple marquant est celui de la puce *Nvidia Tegra X1* qui est intégrée notamment dans la console de jeux *Nintendo Switch*. Un groupe d'attaquants a montré qu'il était possible d'utiliser l'injection de fautes par perturbation de tension afin de sauter l'exécution de certaines instructions. Les attaquants ont ainsi pu révéler le contenu de la mémoire du composant chargé au démarrage de la console, qui contient son *firmware* ainsi que les clefs de chiffrement des autres programmes, cassant ainsi la chaîne de confiance mise en place [63]. Par la suite, cette faille a été exploitée à l'aide d'une attaque par débordement de tampon, permettant à la console de lancer des jeux piratés via l'exploit nommé « Fusée Gelée » [64]. La version suivante de la puce, la *Nvidia Tegra X2* semble aussi être vulnérable à ce schéma d'attaque [61]. Ce système est utilisé dans le milieu automobile avec par exemple, la fonction autopilote de *Tesla* ou encore le panneau de contrôle de *Mercedes-Benz*. On peut légitiment se questionner sur les conséquences de ce genre de failles dans des systèmes aussi critiques, et quelles peuvent être les solutions envisageables pour éliminer de telles vulnérabilités matérielles durant la durée de vie d'un appareil.

1.2.5 Contre-mesures et résilience aux fautes

La résistance aux fautes est un domaine de recherche qui a vu le jour dès les années 1950 avec les premières avancées dans le domaine aérospatial. À cette époque, l'objectif était de concevoir des systèmes électroniques capables de tolérer les fautes d'origine naturelle, telles que celles provoquées par des particules hautement chargées provenant de l'espace. C'est sur cette base que les premiers travaux traitant des injections volontaires de fautes ont été développés. Dans le cadre de cette thèse, nous nous concentrons principalement sur les fautes volontaires, mais les techniques de protection reposent sur des principes de base déjà en œuvre pour lutter contre les fautes naturelles dans les systèmes critiques. Plusieurs contre-mesures peuvent être mises en place pour résister aux fautes, détecter leur manifestation, voire corriger les erreurs induites. Nous débutons par une présentation des contre-mesures matérielles, avant d'aborder les contre-mesures logicielles.

Contre-mesures matérielles

Un bouclier physique comme une plaque de métal peut protéger les puces électroniques contre la lumière ou les perturbations électromagnétiques. Par ailleurs, des capteurs peuvent être intégrés dans le système pour mesurer certaines variables d'environnement et détecter d'éventuelles anomalies [65]. La propagation des signaux d'horloges peut être protégée grâce à des boucles à verrouillage de phase (PLL). Enfin, il est possible d'introduire volontairement de la gigue (*jitter* en anglais), un phénomène de fluctuation de signal, rendant plus difficile la synchronisation de l'injection sans altérer le résultat du programme.

Sur le plan logique, il est possible de dupliquer les calculs de comparer les résultats. En cas de divergence, une erreur est détectée et peut être corrigée à partir de trois résultats (triplication) [65]. La granularité de cette duplication matérielle peut varier : dupliquer un ordinateur complet, un processeur ou seulement l'unité de traitement du processeur sensible aux fautes. Les exécutions peuvent se dérouler avec un décalage temporel pour augmenter la probabilité de détection des erreurs induites par des influences externes. En guise d'exemple, l'organisation lowRISC a publié une implémentation d'un processeur nommée Ibex [66] intégrant une architecture de processeur RISC avec une sécurité augmentée et notamment une configuration en *dual-lockstep* spécifique aux attaques en fautes.

Les mémoires peuvent intégrer des codes détecteurs d'erreur (EDC) ou des codes correcteurs d'erreur (ECC). Ces mesures protègent contre la corruption d'un ou deux bits et s'avèrent efficaces contre les attaques en fautes. Ces mémoires utilisent des bits supplémentaires pour stocker ces codes contenant une redondance de l'information.

De manière complémentaire, certaines solutions combinent des approches matérielles et logicielles. En capturant les signaux à chaque étape du pipeline du processeur, et en les associant aux informations d'exécution d'un programme, il devient alors possible de neutraliser la propagation d'une faute et de garantir l'intégrité du flot de contrôle [67], [68]

Les contre-mesures matérielles nécessitent des modifications ou des ajouts physiques et donc ajoutent un coût supplémentaire important à la production de composants électroniques. De ce fait, l'implémentation de ces contre-mesures est surtout une question de compromis entre le gain en sécurité présumé, la criticité de l'appareil ainsi que son coût de production.

Contre-mesures logicielles

Les contre-mesures logicielles sont introduites dans le code source, lors de la compilation, ou en modifiant le binaire a posteriori. Les premiers travaux se sont concentrés sur la protection des implémentations cryptographiques [32], notamment en remplaçant certains calculs par d'autres équivalents pour renforcer la robustesse des implémentations [69]. Depuis, de nombreuses contre-mesures génériques et applicables à des programmes communs ont été publiées.

De façon analogue aux techniques matérielles, la duplication des opérations dans le programme améliore la robustesse contre les fautes [70]-[72]. Typiquement, une faute impacte une instruction ou une courte séquence d'instructions donc en cas de duplication des opérations, il est probable que seule une opération soit altérée. Ainsi, répéter la même opération plusieurs fois permet de comparer les résultats et de détecter une erreur en cas de divergence. Cette duplication peut se faire à plusieurs niveaux : 1) les instructions manipulant des données peuvent être dupliquées en utilisant des registres différents pour que les opcodes soient distincts ; 2) les appels de fonctions peuvent être exécutés plusieurs fois ; 3) les comparaisons et tests peuvent être réalisés plusieurs fois ; 4) les variables peuvent également être dupliquées. À partir de la triplication, les erreurs peuvent être corrigées directement pendant l'exécution. Les valeurs booléennes peuvent être durcies contre les fautes ciblant les données en modifiant leur encodage. Au lieu d'utiliser la valeur zéro pour une valeur fausse et toutes les autres valeurs pour une valeur vraie comme dans le langage C par exemple, il est préférable d'utiliser deux valeurs distinctes dont la distance de Hamming est maximale et détecter les valeurs en zone blanche.

Afin de s'assurer de l'intégrité du flot de contrôle intra-procédural, des compteurs d'étapes peuvent être intégrés dans le code [73]. Les modifications des compteurs agissent comme des points de passage obligatoires et permettent de détecter un flot de contrôle incorrect.

Un appel de fonction peut être particulièrement sensible, car le compromettre peut permettre à un attaquant de contourner des vérifications. Pour cette raison, il est possible d'utiliser de l'*inlining* de fonction, qui consiste à remplacer l'instruction d'appel par le contenu de la fonction afin d'entrelacer le code et le rendre plus difficile à attaquer.

Les contre-mesures logicielles sont nombreuses et peuvent avoir un lourd impact sur les performances et la taille du programme. De plus, ces protections agrandissent potentiellement la surface d'attaque et peuvent elles-mêmes être la cible d'attaques par injection de fautes. Pour cette raison, on cherche à déployer des contre-mesures efficaces seulement

contre un type de faute ou contre une capacité d'attaquant donné pour limiter les impacts négatifs des mesures de sécurité susmentionnés. Encore une fois, c'est une question de compromis, ici entre le niveau de sécurité et le coût sur les performances [74]. Afin d'ajuster les contre-mesures à intégrer à un programme, il existe des techniques permettant d'optimiser la sélection et le positionnement des mesures de sécurité de façon automatique, pouvant aussi intégrer des protections contre des attaques multi-fautes [75], [76].

1.3 Évaluation de sécurité contre les attaques en fautes

Face à la menace croissante des attaques par injection de fautes, disposer de techniques d'évaluation de sécurité est crucial pour mesurer et renforcer la sécurité des systèmes. Cette section commence par discuter de l'évolution et la faisabilité des attaques multi-fautes. Ensuite, nous passerons en revue quelques méthodes d'analyse de recherche de vulnérabilités actuelles et leurs limites. Enfin, nous abordons les motivations et les objectifs de cette thèse.

1.3.1 Praticabilité des attaques multi-fautes

Avec l'amélioration des bancs d'injection de fautes, il est désormais possible d'injecter plusieurs fautes lors d'une même attaque. De surcroît, ces fautes sont devenues plus précises localement et temporellement, augmentant le risque associé aux attaques [4]-[10]. Cette évolution est relativement récente et ouvre la voie à de nouvelles attaques où les effets de fautes se combinent afin de créer des exploitations plus complexes. Au cours des dernières années, la communauté scientifique autour des problématiques liées à la sécurité physique des appareils s'est penchée sur ce type d'attaques émergentes. La figure 1.4 illustre l'intérêt croissant que représentent les attaques multi-fautes dans le temps dans de nombreux sujets comme les techniques d'injection, d'exploitations ou encore d'analyse de vulnérabilités. L'abscisse représente les années et les ordonnées donnent le nombre d'articles de la littérature mentionnant le terme « multi-faute » dans leurs introductions ou leurs résumés. À noter que ces nombres intègrent probablement des articles en lien avec la résistance aux fautes naturelles. Nous constatons que l'intérêt pour les attaques multi-fautes a commencé à croire à partir de l'année 2016 qui tend à augmenter avec les années.

Afin de se protéger contre les attaques par injection de fautes, de nombreuses contre-

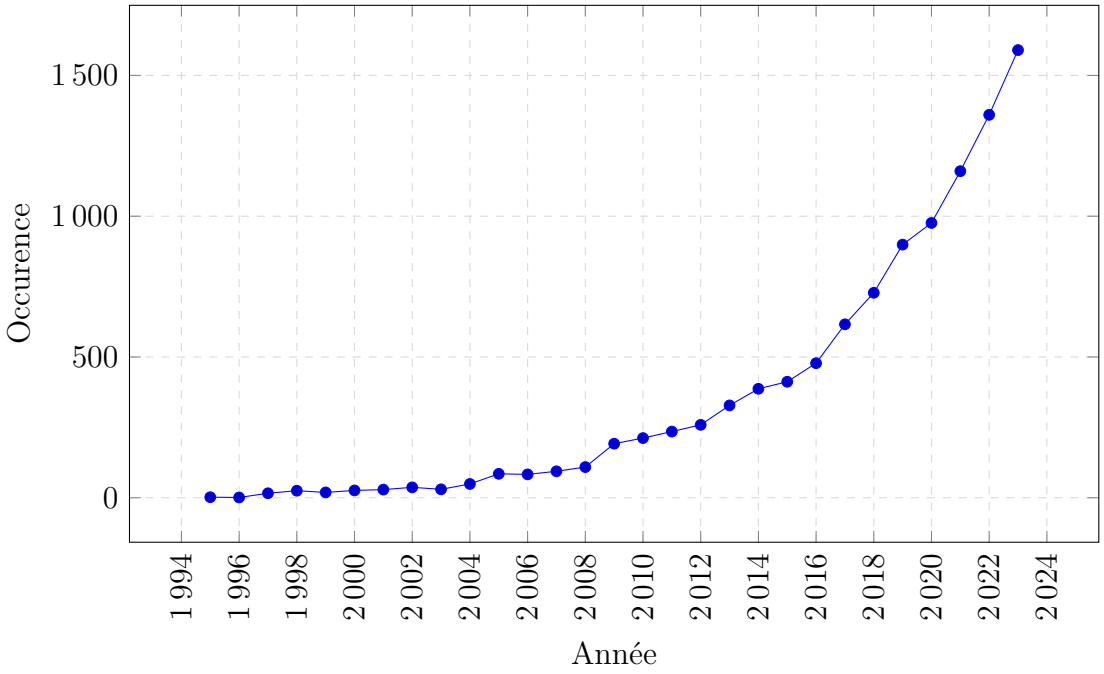


FIGURE 1.4 – Nombres d'articles ayant le terme « multi-faute » dans son résumé ou son introduction dans la littérature selon les années d'après les données de Google Scholar.

mesures logicielles ont été proposées. Ces protections reposent essentiellement sur le principe de redondance [2] : les vérifications ou calculs sensibles sont dupliqués, les valeurs constantes sont encodées de telle sorte qu'il soit difficile de modifier leurs valeurs de façon cohérente. Des variables supplémentaires peuvent également être ajoutées au code afin de surveiller le flot de contrôle et vérifier sa validité par rapport au programme original [3]. En conséquence, il est dorénavant nécessaire d'injecter plusieurs fautes, pouvant impacter potentiellement plusieurs instructions, afin de contourner les contre-mesures mises en place et atteindre les objectifs d'une attaque. Dans un contexte où un attaquant est en mesure d'injecter avec précision de nombreuses fautes, ces contre-mesures logicielles peuvent être ciblées et compromises, ce qui remet en question la sécurité des systèmes jusqu'alors considérés comme résiliants [77]. À partir de ce constat, Péneau et al. [78] dressent un modèle théorique où un adversaire serait capable de sauter l'exécution (*instruction-skip*) de n'importe quelles instructions lors d'une exécution d'un programme. Les auteurs démontrent que cette capacité permet de maîtriser le flot de contrôle et d'effectuer diverses actions, telles que la manipulation des appels de fonctions et de leurs paramètres, la régulation du nombre d'itérations d'une boucle, ou encore l'altération des données en mémoire. Ce

modèle est tellement permissif que si le programme contient un nombre suffisant d'instructions, il est alors complet au sens de Turing. Bien que les moyens d'injection de fautes actuels ne permettent pas ce type d'attaque théorique, nous pouvons faire la supposition que les techniques d'injection continueront de se sophistiquer et que la menace est toujours croissante. En réponse à l'évolution de ces risques, il est nécessaire d'avoir des outils et méthodes d'évaluation de la sécurité capable de prendre en compte l'évolution technique des attaquants.

1.3.2 Détermination des vulnérabilités

Un chemin d'attaque décrit les différentes étapes nécessaires pour réaliser un exploit, incluant les portions de programme à exécuter et les fautes à injecter. Identifier ces chemins permet de mesurer la vulnérabilité d'un système face à des exploitations potentielles, ainsi que d'évaluer et d'améliorer ses contre-mesures. Cependant, la recherche manuelle de vulnérabilités est une tâche fastidieuse, voire impossible dans le contexte des attaques multi-fautes et de la combinaison des effets possibles. C'est pourquoi des outils d'analyse ont été développés pour automatiser, au moins en partie, la recherche de chemins d'attaque. Ces outils d'analyse fournissent une aide précieuse aux évaluateurs et concepteurs de sécurité. Ceux-ci se basent sur un modèle d'attaquant, défini d'une part par le type d'exploit considéré et d'autre part les capacités de l'attaquant à injecter des fautes, notamment leur nombre, leur précision et la nature des effets induits.

Plusieurs approches ont été étudiées dans la littérature et se basent majoritairement sur trois techniques :

— La simulation [12], [13] exécute le programme avec des valeurs d'entrées concrètes en intégrant des fautes afin d'observer la présence de vulnérabilités exploitables par faute. De nombreuses exécutions sont nécessaires pour couvrir toutes les entrées possibles, les fautes et l'ensemble des flots de contrôle permis par le programme.

— L'exécution symbolique [14], [15] représente les paramètres d'entrée, les valeurs des variables et les prises de branchements sous formes de formules logiques qui sont ensuite analysées à l'aide de solveur pour connaître si la présence de fautes peut mener à des attaques.

— La vérification de modèle (ou *model-checking* en anglais) [16]-[18] crée un modèle du programme et vérifie mathématiquement que certaines propriétés de sécurité sont respectées même en présence de fautes.

La plupart des outils existants rencontrent des difficultés à déterminer des chemins d'attaque en présence de fautes précises et nombreuses. Vérifier exhaustivement les effets des fautes ou utiliser des solveurs de formules entraîne une explosion combinatoire lors de l'analyse, rendant ces outils incapables de passer à l'échelle, tant en termes de nombre de fautes que de taille de code considéré.

1.3.3 Objectifs de cette thèse

Lors du démarrage de cette thèse en 2021, seulement quelques solutions d'analyse de sécurité considéraient les attaques multi-fautes et avaient des limites fortes comme un nombre très réduit de fautes ou la capacité d'analyser uniquement des codes courts. Cette situation a motivé cette thèse où nous avons cherché à explorer une nouvelle approche permettant de passer à l'échelle pour des attaques comprenant des fautes précises et nombreuses. Dans un premier temps, nous proposons une méthode d'analyse de programme au format binaire qui s'appuie sur des techniques d'analyses statiques et des heuristiques.

Typiquement, les analyses de sécurité expriment les chemins d'attaque au niveau logiciel, c'est-à-dire qu'elles identifient quelles sont les instructions et les variables devant être fautées pour réaliser l'attaque souhaitée. Toutefois, la mise en œuvre pratique de ces attaques est toujours difficile et donc réservée aux experts. Les bancs d'injection requirent des paramètres de nature physique tels que le moment et la durée des injections exprimés en nombre de cycles ou millisecondes, ou encore la position et l'intensité de l'injection. Dans un deuxième temps, cette thèse propose une méthode permettant de convertir des chemins d'attaque logiciels en paramètres physiques d'injection de fautes, ainsi qu'une technique permettant d'automatiser la réalisation des injections pour une plateforme spécifique et le traitement des résultats des attaques.

1.4 Conclusion

La sécurité d'un système peut être compromise par des attaques exploitant des vulnérabilités logicielles, microarchitecturales ou matérielles. Cette thèse se concentre spécifiquement sur les attaques par injection de fautes, qui consistent à introduire une perturbation matérielle pendant l'exécution d'un programme, entraînant des modifications du flot de contrôle. Avec une précision suffisante, un attaquant peut exploiter ces fautes pour réaliser des attaques complexes. Les fautes peuvent être induites par de nombreux moyens : lu-

mière focalisée, impulsions électromagnétiques, manipulation du signal d'alimentation ou contrôle du signal d'horloge. Après l'introduction de la perturbation dans la cible, la faute se propage depuis le niveau physique jusqu'au niveau logique, se manifestant finalement au niveau logiciel. Ces effets peuvent se traduire par des modifications du flot de contrôle ou des données.

Les techniques d'injection évoluent et permettent désormais d'introduire plusieurs fautes avec précision au cours d'une même attaque [5]-[10]. Ces attaques multi-fautes sont particulièrement redoutables, car elles peuvent contourner les contre-mesures existantes et mener à des attaques plus complexes. Pour répondre à ces nouvelles menaces, il est nécessaire de disposer de méthodes d'évaluation de sécurité permettant d'identifier des vulnérabilités d'un système face à des fautes multiples. Le chapitre suivant introduit les principes généraux de l'analyse de programme, avant de présenter l'état de l'art des techniques d'analyse d'attaques en fautes.

Chapitre 2

État de l'art sur l'analyse de vulnérabilité contre les attaques par injection de fautes

Face à la menace croissante liée aux attaques par injection de fautes, il est devenu essentiel de disposer de méthodes permettant d'évaluer le niveau de vulnérabilité des systèmes. Ces méthodes d'évaluation doivent s'adapter aux évolutions des plateformes d'injection de fautes et plus spécifiquement aux risques relatifs aux attaques multi-fautes. Les analyses de vulnérabilité permettent de déterminer les attaques potentielles, sans nécessiter de réaliser de campagnes d'injections physiques. De ce fait, elles forment une aide précieuse aux évaluateurs de sécurité et concepteurs de contre-mesures.

Dans un premier temps, les concepts généraux de l'analyse de programme sont introduits dans la section 2.1. La section 2.2 propose ensuite un état de l'art des approches d'analyse pour les attaques par injection de fautes. Dans la section 2.3, nous énonçons les objectifs de cette thèse, répondant à des limites identifiées dans les travaux existants.

2.1 Concepts d'analyse de programme

Cette section présente les concepts fondamentaux liés à l'analyse de programme, utilisés par les travaux de l'état de l'art. Nous commençons par formaliser l'ensemble des états d'un système en présence de fautes, ainsi que l'explosion combinatoire liée au nombre de fautes à considérer par l'analyse. En conséquence, les analyses utilisent des approximations pour extraire des informations sur les comportements du programme. Nous définissons les différents niveaux d'approximation possibles et terminons par une classification des méthodes d'analyse de programme.

2.1.1 États d'un système en présence de faute

Une analyse de programme permet d'obtenir des informations sur le comportement d'un programme ou sur la validité de certaines propriétés. Un programme peut être modélisé sous la forme abstraite d'un système de transition d'états, contenant des états dont certains sont initiaux et où les transitions entre ces états représentent des actions issues de l'exécution du programme [79]. Dans notre contexte, les injections de fautes peuvent aussi être représentées comme des actions. Un état est représenté en fonction du niveau d'abstraction choisi pour l'analyse. Au niveau ISA par exemple, un état peut être défini comme les valeurs des registres, le contenu des différentes mémoires, les adresses de l'instruction actuelle et de la suivante. Quant aux actions, elles peuvent être l'exécution nominale d'une instruction du programme (en langage C ou assembleur), ainsi que l'application des effets d'une faute selon un modèle de faute donné. En représentant de manière abstraite les comportements possibles d'un système, une propriété de sécurité peut être formalisée et l'analyse de programme permet d'en vérifier la validité ou non, selon les états initiaux du système.

Comme présenté dans le chapitre 1, il existe une variété de modèles de fautes, pouvant impacter les données manipulées ainsi que le flot de contrôle. Selon les modèles pris en compte par l'analyse, le nombres d'états accessibles par le programme peut croître sensiblement, car au moins une transition est induite pour chaque injection réalisée. Cependant, ce sont surtout les avancées récentes des plateformes d'injection, désormais capables d'injecter plusieurs fautes, qui font exploser le nombre d'états. Les potentielles combinaisons des actions liées aux fautes entraînent une augmentation exponentielle du nombre d'états accessibles, en fonction du nombre de fautes considéré. Cette explosion combinatoire complique l'analyse, parce qu'il devient impossible d'explorer exhaustivement tous les états possibles. Des approximations sont donc nécessaires pour mener à bien l'analyse du système en présence de fautes.

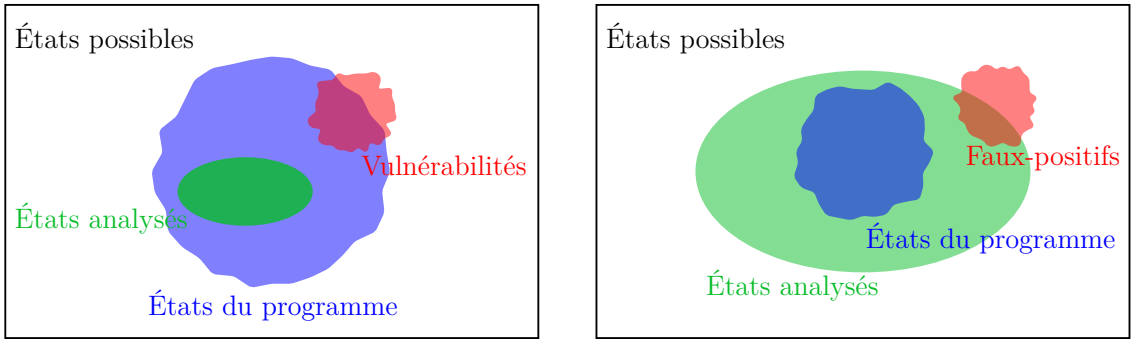
2.1.2 Approximation dans les analyses

Quelle que soit la méthode d'analyse employée, celle-ci repose nécessairement sur des approximations, soit en simplifiant son modèle, soit en limitant le nombre d'états à considérer. Le théorème de Rice énonce que toute propriété non triviale d'un programme est indécidable [80]. Une propriété est définie comme non triviale si elle n'est ni vraie ni fausse pour toutes les fonctions calculables. Par conséquent, de nombreuses méthodes ont été

adoptées pour réaliser des analyses en contournant ces limitations. Ces méthodes visent à réduire la taille du modèle de l'analyse tout en conservant une représentation fidèle de la réalité du programme.

Avant d'examiner les différents types d'approximation et leurs techniques associées, il est nécessaire de définir deux concepts clefs relatifs aux résultats d'analyse. On considère par la suite une analyse de programme retournant un ensemble de preuves pour une propriété donnée. Ces preuves peuvent être vraies ou fausses. Premièrement, la complétude (*completeness* en anglais) désigne la capacité d'une analyse à retourner toutes les preuves qui sont vraies pour une propriété donnée, garantissant ainsi l'absence de faux-négatifs. Deuxièmement, la correction (*soundness* en anglais) désigne la capacité d'une analyse à retourner des preuves qui sont uniquement vraies, ne produisant ainsi aucun faux-positif.

Dans le cadre de l'analyse de programme, plusieurs niveaux d'approximation existent, permettant de collecter des informations utiles pour l'évaluateur ainsi que pour prouver des propriétés de sécurité ou de robustesse d'un système. Pour formaliser ces approximations, nous définissons d'abord l'ensemble des états possibles, c'est-à-dire tous les états qui existent dans le modèle formel utilisé par l'analyse. Au sein de cet ensemble, nous distinguons le sous-ensemble des états atteignables par le programme, potentiellement en présence de fautes, ainsi que le sous-ensemble des états explorés par l'analyse.



(a) Sous-approximation (b) Sur-approximation

FIGURE 2.1 – Illustration des types d'approximation

Sous-approximation

L'analyse explore un sous-ensemble des états du programme. Cette approximation est illustrée par la figure 2.1a, où l'ensemble des états du programme est représenté en bleu, tandis que l'ensemble des états couverts par l'analyse est en vert. De fait, certaines vulné-

rabilités peuvent échapper à l'analyse, comme le montre l'intersection entre les états du programme en bleu et les vulnérabilités en rouge. Cette approche rend l'analyse correcte, mais en revanche pas complète. Les analyses basées sur des techniques de simulation ou de *fuzzing* s'appuient sur la sous-approximation, car elles n'explorent qu'une portion des exécutions possibles d'un programme.

Sur-approximation

L'analyse couvre l'ensemble des états du programme et aussi d'autres états inaccessibles par le programme. La figure 2.1b illustre cette approximation, où l'ensemble des états du programme et l'ensemble des états analysés sont respectivement représentés en bleu et en vert. Les analyses s'appuyant sur la sur-approximation peuvent identifier des faux-positifs, comme représenté par l'intersection entre les vulnérabilités en rouge et les états couverts par l'analyse en vert. Par conséquent, cette approche rend l'analyse complète, mais potentiellement pas correcte. Des moyens de filtrage peuvent être mis en place pour pallier le problème de faux-positifs. Par exemple, l'interprétation abstraite [81] repose sur ce type d'approximation.

Approximation hybride

Il existe des analyses reposant sur une approximation hybride, couvrant à la fois un sous-ensemble des états accessibles du programme ainsi que des états inaccessibles. En conséquence, ce type d'analyse peut manquer des vulnérabilités et générer des faux-positifs, ce qui signifie que cette approche n'est ni complète ni correcte. Les techniques d'exécution concolique [82] utilise ce type d'approximation.

2.1.3 Catégorisation des techniques d'analyse

La littérature couvre une vaste gamme de techniques d'analyse de programme. L'ensemble des états possibles pour un programme donné peut être exploré partiellement pour identifier des chemins d'attaque, ou de manière exhaustive pour prouver la robustesse du système. Les effets des fautes résultent de l'application d'un modèle de faute sur un état du système, engendrant de nouveaux états accessibles. Il existe deux principales catégories d'analyse : les analyses dynamiques et les analyses statiques.

Les analyses dynamiques consistent à exécuter le programme pour explorer les états possibles et étudier son comportement. Par opposition, les analyses statiques n'exécutent

pas le programme et se basent uniquement sur sa description sémantique pour appliquer des raisonnements mathématiques. Cette description sémantique peut varier selon le niveau d'abstraction requis par le modèle de faute : un code source, des instructions issues d'un programme binaire, ou encore une description de la microarchitecture au niveau RTL avec des fichiers VHDL ou Verilog.

À partir de cette description, une représentation intermédiaire est éventuellement générée, sur laquelle des outils mathématiques peuvent être appliqués. Plusieurs approches existent pour prouver des propriétés de sécurité. Une méthode consiste à formuler les comportements du programme sous forme de formules propositionnelles, c'est-à-dire des expressions booléennes portant sur les variables, permettant ainsi de raisonner sur les implications des affectations logiques. La logique du premier ordre, qui inclut la logique propositionnelle, introduit l'usage des quantificateurs et des prédicats sur les éléments du domaine (variables et symboles) [83]. Le calcul de prédicats permet ainsi de modéliser symboliquement des ensembles d'exécutions possibles du programme [84]. Pour vérifier la validité de propriétés formelles, des solveurs peuvent être utilisés, notamment des solveurs SAT [85] pour la logique propositionnelle et les solveurs SMT [86] pour les calculs de prédicats.

La figure 2.2 représente une catégorisation des différentes techniques d'analyse et la façon dont sont modélisés les états ainsi que la méthode pour les explorer. Nous pouvons constater que les frontières entre les différentes techniques sont fines et que de nombreuses solutions utilisent une combinaison de méthodes. Par conséquent, il est parfois impossible de ranger dans une seule catégorie de nombreux outils présents dans l'état de l'art.

Les analyses raisonnent sur les transitions entre les états possibles du programme. Ces états intègrent de l'information en fonction du niveau d'abstraction de l'analyse et du modèle de faute considéré. Dans le cadre de recherche de vulnérabilité, chacun de ces niveaux possède des contraintes. En guise d'exemple, l'analyse au niveau du code source ne peut pas prendre en compte l'agencement des instructions du binaire. Il en va de même pour l'analyse au niveau ISA qui ne prend pas en compte l'état de la microarchitecture, en plus de nécessiter un désassembleur [87].

2.2 Outils et méthodes issus de la littérature

De nombreux outils et méthodes ont été publiés dans la littérature et beaucoup d'entre eux combinent plusieurs approches d'analyse. Cette section présente un état de l'art de

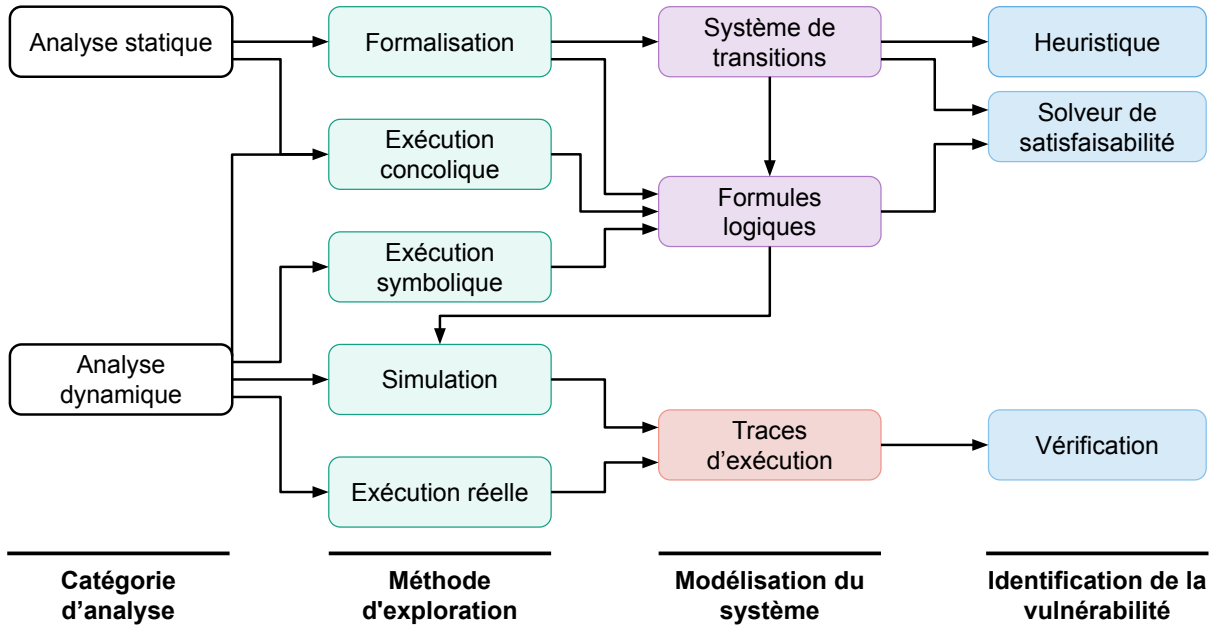


FIGURE 2.2 – Catégorisation des analyses de programme

ces travaux en les classant en trois catégories : la simulation, l'exécution symbolique et la vérification de modèles. Chaque solution est classée en fonction de la technique principale employée.

2.2.1 Simulation

La simulation est une technique d'analyse dynamique de programme qui consiste à exécuter le programme avec des valeurs d'entrée concrètes, potentiellement en présence de fautes, pour observer le comportement résultant. Le simulateur permet de suivre l'évolution de l'état du système pendant l'exécution afin de déduire l'impact des paramètres et des effets des fautes. L'usage de valeurs concrètes signifie que l'analyse ne considère qu'un seul chemin d'exécution à la fois, contrairement aux analyses basées sur l'exécution symbolique. La principale difficulté de cette technique réside dans le choix des entrées et des injections de fautes, pour maximiser la couverture des chemins d'exécutions et scénarios de fautes.

Au niveau microarchitectural

En 1994, Jenn et al. sont les premiers à proposer une technique de simulation de faute avec l'outil *MEFISTO* [88]. L'article décrit une méthodologie de simulation de faute au

niveau de la microarchitecture en se basant sur le langage de description de matériel *VHDL*, tout en nécessitant peu, voire pas de modification du code de base. Le VHDL décrit le comportement des signaux entre les composants du processeur et les variables qui y sont manipulées. *MEFISTO* permet d'introduire des composants saboteurs qui vont introduire des perturbations dans la transmission des signaux ou des composants mutants qui sont des versions modifiées d'un composant existant permettant d'injecter des fautes. Les fautes introduites peuvent être temporaires ou permanentes selon le scénario de la simulation. Les expérimentations ont été menées sur un processeur 32-bits exécutant des algorithmes de tri et ont permis d'identifier plusieurs vulnérabilités. Ces travaux montrent les mécanismes de propagation de fautes, du signal compromis jusqu'au niveau logiciel.

Au niveau ISA

Machemie et al. proposent l'outil *SmartCM* [89], visant à analyser l'effet des fautes sur les programmes des cartes à puces basées sur le système d'exploitation Java Card. Leur méthode consiste à créer des mutants en modifiant les valeurs de la mémoire morte de la carte (EEPROM) contenant le programme et ses données. Les mutants qui ne sont pas détectés par les contre-mesures mises en place sont analysés. Ceux qui sont identifiés comme potentiellement dangereux sont simulés afin d'observer leurs comportements. Le modèle de faute considéré est le *set* ou le *reset* d'un octet précis en mémoire qui consiste à affecter respectivement et de façon permanente la valeur 0x00 ou 0xFF. Si la mémoire est chiffrée, la valeur de l'octet est aléatoire et sans contrôle pour l'attaquant. La recherche de mutant est exhaustive et ne permet de considérer seulement qu'une faute pendant l'attaque. Leurs expérimentations montrent la capacité de leur approche à évaluer le niveau d'efficacité de plusieurs contre-mesures logicielles implémentées dans la machine virtuelle Java.

Schirmeier et al. proposent une infrastructure logicielle nommée *FAIL** [90] permettant de mesurer le niveau de tolérance aux fautes d'un système. Contrairement aux autres solutions, *FAIL** fournit un environnement qui automatise la préparation des simulations, leurs exécutions ainsi que le traitement et la visualisation des résultats. La simulation est réalisée à partir de simulateurs matériels ou d'outils d'instrumentation existants, tels que *Bochs*, *Gem5* ou encore *OpenOCD*. *FAIL** propose de nombreux modèles de fautes issus de la littérature, notamment sur des modifications de données en mémoires ou contenues dans les registres, mais il est possible d'intégrer d'autres modèles personnalisés à partir d'une bibliothèque en C++. L'infrastructure emploie une architecture client-serveur afin

de distribuer efficacement le temps de calcul lié à la préparation des tests à réaliser puis des simulations.

Dureuil et al. proposent *CELTIC* [12], un simulateur qui se concentre exclusivement sur les effets des fautes identifiées comme réalisables pour une carte embarquée et un équipement d'injection spécifique. À partir des résultats issus d'une étape préalable de caractérisation des fautes en utilisant un banc d'injection électromagnétique et laser, des modèles de fautes et leurs taux de reproductibilité sont manuellement inférés. Ces modèles approximent les effets des fautes sur les données en mémoire, mais ne prennent pas en compte les modèles relatifs à la modification d'instruction. *CELTIC* utilise ces modèles pour simuler exhaustivement les potentielles attaques et fournit un score de vulnérabilité qui inclut également la faisabilité des fautes. Cette approche permet de réduire le nombre d'exécutions à simuler pour des scénarios d'attaques impliquant une seule faute.

Werner et al. [13] proposent une amélioration de l'approche de Dureuil et al. [12], en introduisant une méthode permettant d'inférer automatiquement les modèles de fautes et permet de réaliser des attaques combinant deux fautes. Les auteurs montrent l'intérêt de l'utilisation de modèles de fautes spécialisés pour la plateforme d'injection employée et le système ciblé par rapport à des modèles génériques. Pour trouver les paramètres d'injection les plus intéressants et limiter la quantité de tests à réaliser, la caractérisation utilise une stratégie de recherche par grille et un banc d'injection laser comprenant deux sources. La méthode intègre *CELTIC* afin d'inférer plus efficacement les modèles de fautes spécifiques à la cible. La recherche de chemins d'attaque est toujours exhaustive et ne permet pas de considérer les attaques avec de nombreuses fautes. Les expérimentations démontrent l'efficacité de l'outil sur plusieurs binaires, mais se limitent à seulement deux fautes. De plus, les auteurs supposent que les fautes ne dépendent pas du code exécuté, mais seulement des paramètres d'injection. Dans ce cadre, les attaques multi-fautes sont envisagées comme de simples combinaisons d'attaques à une seule faute, sans prendre en compte l'ensemble des effets qu'une faute pourrait avoir sur le contexte d'exécution et donc des influences sur les fautes suivantes.

Hoffmann et al. présentent une solution combinant un simulateur au niveau ISA, appelé *M-ulator*, dédié à l'architecture Arm, permettant de simuler les effets des fautes et une technique d'automatisation des simulations, nommée *ARMORY*, pour évaluer la sécurité d'un programme binaire [91]. *ARMORY* prend en charge plusieurs modèles de fautes, tels que la modification d'instructions (remplacement ou saut) et l'altération des données des registres. L'identification des vulnérabilités repose sur une exploration exhaustive de

toutes les positions de fautes possibles. La simulation s'arrête à des points d'arrêt définis par l'évaluateur et l'attaque est validée si l'état du système correspond à la spécification de l'exploit. *M-ulator* permet de simuler des attaques multi-fautes, mais la complexité exponentielle due à l'explosion combinatoire des effets des fautes rend difficile l'analyse pour des scénarios d'attaques multi-fautes. Les auteurs appliquent *ARMORY* sur une implémentation de l'algorithme AES et d'un programme de démarrage sécurisé. Cette approche exhaustive limite les attaques à deux fautes. Néanmoins, les résultats démontrent l'influence des optimisations de compilation et de l'agencement du binaire sur la quantité de vulnérabilités identifiées, pour un même programme. Ils expliquent que des contre-mesures conçues à un niveau d'abstraction élevé, comme le code source, ne montrent pas les problématiques d'agencement du code ou le choix des instructions utilisées. Aussi, les auteurs affirment que l'ajout de contre-mesures conçues pour un modèle de faute spécifique peut, paradoxalement, accroître la surface d'attaque pour d'autres types de fautes, créant des vulnérabilités imprévues.

Au niveau du code source

Puys et al. proposent une technique de simulation d'attaques par injection de fautes au niveau du code source en langage C [92]. Deux modèles de fautes sont étudiés : le premier concerne les données, où la valeur d'une variable, y compris temporaire lors d'un calcul, peut être altérée ; le second correspond à un saut de ligne, dans lequel une ligne de code C peut être omise lors de l'exécution. Le simulateur considère les attaques avec une ou plusieurs fautes durant l'analyse, mais de façon exhaustive et doit donc se limiter en cas de multi-faute pour passer à l'échelle. Une implémentation d'un algorithme CRT-RSA est attaquée avec succès et deux contre-mesures logicielles sont démontrées comme vulnérables à ces modèles de faute.

2.2.2 Exécution symbolique

L'exécution symbolique est une technique d'analyse où le programme est exécuté avec des entrées et paramètres symboliques plutôt qu'avec des valeurs concrètes [84]. Ces valeurs symboliques représentent des ensembles ou des plages de valeurs possibles. Au cours de l'exploration du programme, l'analyse construit progressivement des chemins d'exécution avec des expressions logiques modélisant les contraintes sur les entrées et les éventuelles injections de fautes à satisfaire pour chaque chemin exécuté. À la fin, on obtient

un système de contraintes souvent complexe, nécessitant l'utilisation de solveurs de satisfaisabilité (SAT [85] ou SMT [86]) pour identifier des vulnérabilités dans l'ensemble des chemins d'exécution.

Analyse de robustesse

Larsson et al. ont été les premiers à proposer une solution utilisant l'exécution symbolique [93]. Leur outil permet d'analyser du code Java et d'identifier les vulnérabilités en présence de fautes altérant le contenu des variables. Le modèle de faute considéré est le *bit-flip*, affectant un ou plusieurs bits. Avant de lancer l'analyse, des pseudo-instructions sont intégrées dans le code source pour déclencher les effets des fautes sur l'état symbolique lors de l'exécution. Certains invariants, comme le nombre d'itérations des boucles, sont renseignés manuellement. L'outil utilise le moteur d'exécution *KeY*, combiné avec des règles spécifiques aux fautes, pour réaliser des preuves de robustesse de manière semi-automatique. Bien que ces travaux soient initialement destinés aux fautes naturelles, ils ont démontré la faisabilité de l'exécution symbolique dans l'analyse de sécurité liée aux fautes.

Approche par bifurcation (*forking*)

L'approche par bifurcation (*forking*) consiste à modéliser le programme analysé avec des points de bifurcation (c'est-à-dire des branches) aux éventuelles positions des fautes. Ces bifurcations sont contrôlées par des variables booléennes, indiquant si une faute spécifique est activée ou non lors de l'exécution. Le nombre maximal de fautes peut être borné pour garantir un passage à l'échelle. Contrairement aux techniques utilisant la génération de mutants, des portions de l'analyse peuvent être mises en commun avec plusieurs chemins d'exécution et activations de fautes.

Cette approche a été utilisée en premier par Potet et al., en proposant *Lazart* [14], un outil d'analyse de robustesse contre les attaques multi-fautes affectant le flot de contrôle d'un programme. Le type d'exploit considéré est celui de l'accessibilité, c'est-à-dire la capacité d'un attaquant à atteindre certaines sections du code tout en évitant d'autres, afin de réaliser des actions non-permises. Le modèle de faute utilisé par *Lazart* est l'inversion de test, permettant à un attaquant de modifier le résultat d'un saut conditionnel pour prendre la branche désirée. Ce modèle de haut niveau peut résulter de fautes de plus bas niveau, comme le saut d'instruction pour contourner une instruction de saut ou une affectation des indicateurs de retenue (*carry flags*), ou encore l'altération de bits modifiant

les données et validant le branchement souhaité. Pour modéliser les effets potentiels des fautes, *Lazart* utilise l'analyse statique pour reconstituer le graphe de flot de contrôle initial du programme, puis génère un seul mutant d'ordre supérieur, contenant une variable booléenne pour tous les sauts conditionnels permettant de décider de la branche qui sera prise. À partir de ce graphe, *Lazart* utilise un algorithme de coloration pour calculer les propriétés d'accessibilité aux sections du code visées. Enfin, ce graphe coloré est exploité pour déterminer des chemins réalisant l'exploit selon un nombre de fautes prédéfini. Les auteurs montrent la capacité de l'outil à trouver des vulnérabilités nécessitant jusqu'à cinq fautes dans un temps relativement réduit. Grâce à l'approche par bifurcation combinée au moteur d'exécution *KLEE* et à sa stratégie de couverture de code, *Lazart* parvient à passer à l'échelle pour l'analyse multi-faute. Toutefois, cette approche présente certaines limitations, notamment dues à son niveau d'abstraction. *Lazart* se base sur le bytecode *LLVM*, une représentation intermédiaire utilisée par le compilateur lors de la transformation du code source en assembleur pour une architecture donnée. Ce niveau d'abstraction peut être trop éloigné de l'agencement réel du programme binaire : le saut d'une ligne dans cette représentation peut correspondre à plusieurs instructions, qui ne sont pas nécessairement contiguës. Cette représentation est suffisante pour représenter le modèle de faute de l'inversion de tests et a l'avantage de rendre l'analyse compatible avec plusieurs langages de programmation. Dans certains cas, l'analyse prend plus de temps que prévu et est automatiquement interrompue. La robustesse du code ne peut alors être prouvée et le résultat est classé comme non concluant. Malgré ces quelques points, ces travaux ont été les premiers à gérer l'explosion combinatoire des fautes et ont rendu possibles beaucoup d'autres travaux d'analyse de sécurité.

Approche par encodage de fautes

Les techniques mentionnées précédemment ne passent pas à l'échelle dans un contexte d'attaques avec des fautes nombreuses, car le nombre de mutants générés ou de chemins explorés explose avec le nombre de fautes considérées. En réponse à ce problème, Ducousso et al. [15] proposent une approche basée sur l'encodage des fautes, ne nécessitant pas de bifurcation (*forkless*), et permettant d'identifier des exploits d'accessibilité. Leur solution, appelée *FASE*, encode les fautes sous la forme d'une variable booléenne représentant l'activation de la faute et son effet sur l'état du système. Cet encodage permet de supporter plusieurs modèles de fautes : la manipulation de bits à différentes granularités (bit, octet et mot) et l'inversion de tests, puis le saut d'instruction dans la deuxième version

présentée dans la thèse de Ducousso [94]. L'encodage permet aussi de positionner simultanément toutes les fautes possibles, puis de sélectionner les fautes les plus pertinentes pour réaliser un exploit donné. Le nombre maximal de fautes actives en même temps permet de gérer les attaques multi-fautes. Cette approche résout le problème de l'explosion combinatoire du nombre de chemins, mais au prix de l'augmentation de la complexité des expressions symboliques à calculer. Pour diminuer cette complexité, deux optimisations sont proposées, permettant de réduire le nombre de fautes nécessaires pour un chemin donné et de simplifier les formules manipulées. Une implémentation de *FASE*, basée sur l'infrastructure de développement *BINSEC*, est utilisée lors des expérimentations et opère sur les ISA des architectures x86 et Arm. Les résultats montrent que *FASE* est efficace pour identifier des chemins d'attaque multi-fautes, comprenant jusqu'à dix instructions fautées au maximum. Ces travaux démontrent la pertinence de cette approche, faisant de *FASE* l'un des outils les plus performants de l'état de l'art.

2.2.3 Vérification de modèles

La vérification de modèles (ou *model checking* en anglais) est une technique permettant de vérifier si un modèle satisfait une propriété donnée. Le modèle est un système de transitions d'états, représenté par un graphe orienté qui abstrait les différents états du programme ainsi que les transitions entre ces états. Ce graphe est construit à partir de la description sémantique du programme, comme le code source ou les instructions issues d'un binaire. Dans le contexte de l'analyse de sécurité contre les attaques par injection de fautes, les propriétés à prouver incluent la résistance du système face à un modèle de faute défini, avec un nombre spécifique de fautes. De nombreux algorithmes permettent d'explorer les états possibles du système et de prouver ces propriétés, tels que les solveurs de satisfaisabilité, qui peuvent prouver l'absence de trace menant à un état de vulnérabilité.

Approche par génération de mutants

L'approche par génération de mutants consiste à analyser des versions modifiées d'un programme, appelées « mutants », où chacun intègre les effets d'une ou plusieurs fautes spécifiques. L'objectif est d'observer les impacts éventuels des fautes en comparant les exécutions des mutants à celui du programme original. Given-Wilson et al. proposent une méthode d'analyse de sécurité basée sur la génération de mutants [95]. Dans cette ap-

proche, les propriétés de sécurité sont d'abord spécifiées directement dans le code source sous forme d'annotations. À partir du binaire du programme intégrant ces propriétés, des mutants sont générés en appliquant exhaustivement les effets des fautes. Les modèles de fautes pris en compte incluent le *bit-flip*, la réinitialisation de données à différentes granularités (bit, octet, mot), le saut d'instruction et l'inversion de tests. Une pré-analyse est effectuée pour filtrer les mutants qui plantent lors de l'exécution, tandis que les mutants viables sont ensuite analysés sous forme de représentation intermédiaire LLVM. Les validations des propriétés des mutants sont ensuite comparées à celles du programme original. En cas de différence, une potentielle vulnérabilité est identifiée, nécessitant une analyse manuelle pour confirmer l'attaque. Cette approche a permis de découvrir des vulnérabilités dans deux implémentations d'algorithmes de chiffrement. Cependant, la génération exhaustive de mutants reste limitée par l'explosion combinatoire du nombre de mutants lorsque des attaques multi-fautes sont considérées.

Au niveau microarchitectural

Tollec et al. proposent l'outil *μArchiFI* [18], qui vise à identifier des attaques multi-fautes au niveau microarchitectural. Cet outil est basé sur *Yosys* et utilise une modélisation formelle prenant en compte la propagation des fautes depuis le niveau matériel jusqu'au niveau logiciel. La vérification du modèle intègre un modèle d'attaquant couvrant toutes les positions spatiales et temporelles possibles d'une faute, ainsi que l'ensemble de ses effets. Les modèles de fautes pris en charge par *μArchiFI* sont basés sur la manipulation des bits des registres et de la combinatoire (*bit-flip*). L'outil génère automatiquement un système de transition, où chaque nœud représente les actions du programme et les effets potentiels des fautes. Identifier un chemin d'attaque revient à trouver une suite de transitions menant à un état ciblé par l'attaquant dans le programme. Chaque chemin est représenté par une expression logique, qui est ensuite résolue à l'aide d'un solveur de satisfaisabilité. Afin de réduire la complexité des calculs et le nombre de variables, *μArchiFI* applique une technique de *Sandboxing*, restreignant l'ensemble des valeurs possibles pour le registre du compteur de programme (PC). L'outil prend en compte des scénarios d'attaque impliquant jusqu'à cinq fautes, mais la modélisation de tous les composants du système dans unique modèle pose des problèmes de passage à l'échelle. Par conséquent, *μArchiFI* est limité à une centaine d'instructions pour que le vérificateur de modèles fonctionne.

Afin d'améliorer le passage à l'échelle, Tollec et al. proposent une nouvelle approche [96],

qui s'extrait du modèle monolithique présenté précédemment dans [18]. Cette méthode permet de prouver la robustesse d'un système contre les attaques multi-fautes en tenant compte à la fois du design du processeur au niveau RTL et du programme exécuté. La première étape consiste à prouver formellement la sécurité du circuit, indépendamment du programme, en utilisant la notion de partionnement résistant à k fautes. Une partition représente ici une portion disjointe du circuit du point de vue des registres du système. L'outil évalue la robustesse face à k fautes en s'appuyant sur un solveur de satisfaisabilité. Contrairement à *μArchiFI*, cette vérification matérielle ne doit être réalisée qu'une seule fois et peut être réutilisée pour analyser différents programmes s'exécutant sur le même matériel. Une faute est considérée comme exploitable si les mécanismes de protection ne permettent pas de la détecter. Dans une deuxième étape, un modèle est construit en combinant le binaire du programme, le design du processeur, les fautes exploitables identifiées au préalable et les objectifs de l'attaquant. Des simulations sont ensuite réalisées pour localiser les fautes et des calculs de prédicats sont effectués pour déterminer si ces fautes conduisent à une vulnérabilité. Les auteurs ont utilisé cet outil pour analyser la sécurité du démarrage du processeur *Secure Ibex* [66] sur la puce *OpenTitan*. Cette version a permis d'analyser un programme contenant 2526 instructions, marquant une amélioration significative par rapport aux travaux antérieurs [18].

Au niveau ISA

Pattabiraman et al. proposent *SymPLFIED* [97], un outil pour prouver la robustesse d'un programme contre des attaques en fautes. *SymPLFIED* fonctionne au niveau ISA en basant son analyse sur un jeu d'instructions spécialisé dérivé de *MIPS*. Après avoir collecté tous les états fautés accessibles d'un programme lors d'une exécution symbolique, l'outil utilise des techniques de vérification de modèles pour trouver des chemins d'attaque ou prouver l'absence de vulnérabilité. *SymPLFIED* prend en paramètre les instructions à analyser, un modèle de faute spécifique et un détecteur d'erreur. L'analyse supporte deux modèles de fautes : une faute dans les données qui se traduisent par le remplacement de la valeur contenue dans un registre ou un emplacement mémoire par une variable *err*, tandis qu'une erreur lors de l'exécution d'une opération se manifeste par le remplacement de la valeur d'un registre source ou destination par *err* pendant un calcul ou une lecture en mémoire. Ces modèles de fautes ne prennent pas en compte la granularité d'une faute et *err* peut être la conséquence d'une ou plusieurs altérations de bits. Quant aux détecteurs de faute, ceux-ci sont spécifiés en utilisant du code assembleur et doivent être intégrés au

code. Durant la vérification, tous les états fautés sont exhaustivement considérés et cela engendre par conséquent de nombreux faux-positifs pouvant rendre l'analyse pessimiste.

De nombreuses contre-mesures logicielles ont été proposées pour renforcer la sécurité des programmes contre les attaques par injection de fautes. Cependant, il est crucial de vérifier spécifiquement la robustesse de ces contre-mesures. En réponse à ce besoin, Goubet et al. proposent *RobustA* [98], un outil d'analyse de code au niveau assembleur afin de vérifier l'efficacité des contre-mesures en présence de fautes ayant pour effet un saut d'instruction. Le jeu d'instructions supporté est l'*Arm-Thumb2*. L'outil prend en entrée deux programmes : une version de base et une version sécurisée intégrant des contre-mesures. À partir des traces d'exécution de ces deux programmes, un automate est inféré, où chaque nœud représente une instruction et chaque transition est annotée avec l'action de l'instruction sur l'état du programme et la condition de réalisation. Ce graphe est exhaustivement exploré pour considérer l'ensemble des chemins d'exécution possibles. Ensuite, un solveur SMT est utilisé pour identifier les chemins menant à une attaque réussie. *RobustA* se révèle efficace pour prouver la robustesse dans des scénarios d'attaques avec seulement une faute, mais l'approche est limitée par l'explosion combinatoire en cas d'attaque multi-faute et la taille du code considéré.

Bien que leurs noms soient similaires, Bréjon et al. proposent une approche différente avec leur outil nommé *RobustB* [16], qui combine trois techniques d'analyse. Cet outil permet de trouver des chemins d'attaque et de prouver la robustesse d'un code au niveau ISA pour architecture *ARMv7-M*. *RobustB* se base sur l'infrastructure de développement *angr*. La première étape réalisée par *RobustB* consiste à préparer le modèle pour l'analyse. Le graphe de flot de contrôle du programme est construit à l'aide d'une analyse statique, puis l'ensemble des contextes possibles à l'entrée du premier nœud du graphe est calculée en utilisant de l'exécution symbolique. Durant la seconde étape, tous les chemins accessibles sont exhaustivement listés en dépliant le graphe de flot de contrôle. Un filtre utilisant un solveur SMT élimine les exécutions impossibles. Enfin, la dernière étape vise à identifier si un chemin est vulnérable à une faute en utilisant des expressions logiques et un solveur SMT. Lors de l'analyse, un modèle de faute est utilisé parmi les deux supportés : le saut d'instruction et la corruption de registres où un ou plusieurs bits peuvent être altérés. Les auteurs ont démontré l'efficacité de l'outil pour des attaques avec une seule faute, mais l'utilisation du solveur SMT limite sa capacité à gérer la présence de plusieurs fautes ainsi que la taille du code analysable.

2.3 Problématiques et contributions de la thèse

L'état de l'art propose diverses approches pour évaluer la sécurité des systèmes, mais ces méthodes rencontrent toutes des limitations importantes lorsqu'il s'agit d'analyser un programme en présence de fautes nombreuses. Cette section présente les problématiques et les objectifs de cette thèse, comprenant d'abord une méthode d'analyse de sécurité pour les attaques multi-fautes, puis une méthode de réalisation automatisée d'injections multiples.

2.3.1 Analyse de sécurité pour les attaques multi-fautes

Les attaques par injection de fautes multiples sont puissantes, car elles permettent de contourner les mécanismes de sécurité des logiciels des systèmes embarqués. L'évaluation du niveau de vulnérabilité d'une application face à des injections de fautes multiples, avec des effets pouvant être variés, demeure un problème ouvert en raison de la taille de l'espace de fautes à explorer ou la taille du programme analysé.

Problématique de passage à l'échelle

Seulement une minorité des approches présentées dans la section 2.2 permettent d'évaluer la sécurité d'un programme face aux attaques multi-fautes. Les analyses prenant en compte plusieurs fautes sont toujours limitées à un nombre réduit de fautes permises ou à une taille de code petite. Cela s'explique par les techniques d'exploration des états du système en présence de fautes : le nombre d'états possibles croît de manière exponentielle en fonction du nombre de fautes considéré. Les solveurs de satisfaisabilité et les outils basés sur la simulation cherchent à explorer parfois exhaustivement l'ensemble de ces états. Par conséquent, pour passer à l'échelle, ces approches réduisent la complexité du modèle en fixant un nombre maximal de fautes relativement bas. L'outil FASE, réalisé par Ducousso et al. [15], [94] et publié en 2023, est l'outil supportant le plus grand nombre de fautes, avec jusqu'à dix instructions fautées réparties sur plusieurs fautes. Avec les récentes avancées des plateformes d'injection, nous pensons qu'il est nécessaire de développer de nouvelles approches capables d'analyser des programmes en présence de nombreuses fautes, avec des fautes de largeurs potentiellement variables.

Approche par analyse statique

Le premier objectif de cette thèse est d'explorer la faisabilité d'une approche reposant exclusivement sur l'analyse statique pour répondre aux besoins d'évaluation des programmes dans un contexte d'attaques multi-fautes. Les analyses de programme basées sur l'interprétation abstraite [81] ont déjà pu montrer leur efficacité, notamment pour démontrer des propriétés de sûreté comme pour le projet *ASTREÉ* [99].

Dans le domaine de l'analyse de programme pour l'injection de fautes, l'utilisation de techniques basées sur l'analyse statique reste principalement limitée à la collecte d'informations préliminaires. Par exemple, *FAIL** [90] se sert de l'analyse statique pour réduire le nombre de simulations nécessaires, tandis que *Lazart* [14] et *RobustB* [16] l'emploient pour construire le graphe de flot de contrôle du programme. Également, les travaux de Lacombe et al. [100] et l'outil *SymPLFIED* [97] l'utilisent pour identifier les portions de code les plus pertinentes à analyser. À notre connaissance, aucune méthode n'a été proposée pour analyser la sécurité contre les attaques multi-fautes en se basant exclusivement sur l'analyse statique.

Nous proposons une approche différente, basée sur une méthode d'analyse capable de découvrir des chemins d'attaque réalisables avec un grand nombre de fautes, sans recourir à des techniques d'exploration exhaustive de l'ensemble des états du programme. À la place, des heuristiques sont conçues pour explorer des états fautés encore inaccessibles par les méthodes existantes de l'état de l'art. La contrepartie de l'utilisation d'heuristiques est que notre analyse n'est pas complète donc elle ne permet pas de prouver la robustesse d'un système. Néanmoins, l'identification des chemins d'attaque constitue une aide pour les évaluateurs de systèmes et les concepteurs de contre-mesures. Pour cette analyse, nous nous concentrons sur les fautes induisant des sauts d'instruction avec des largeurs variables. Notre approche, appelée SAMVA, est présentée dans le chapitre 3.

2.3.2 Réalisation de campagnes d'injections multi-fautes

Un processus d'évaluation nécessite, en dernière étape, la réalisation de campagnes d'injection de fautes sur le système visé. Toutefois, dans le cadre d'attaques multi-fautes, la difficulté de ces campagnes augmente avec le nombre de fautes et des techniques permettant d'identifier des paramètres d'attaques sont utiles aux évaluateurs.

Problématique de faisabilité des fautes

Pour des raisons d'efficacité et d'extensibilité, l'analyse est le plus souvent réalisée sur le code binaire au niveau ISA et repose sur des modèles de fautes qui abstraient les effets des injections de fautes à ce niveau. Par exemple, le modèle de faute *instruction-skip* couramment observé au niveau ISA peut résulter de différentes perturbations matérielles, telles qu'une perturbation à l'étape de décodage des instructions du pipeline de l'unité centrale, des inversions de bits dans le codage binaire des instructions ou la relecture d'instructions stockées dans le tampon d'instructions. Cependant, l'effet d'une faute peut considérablement varier en fonction des conditions de son injection comme l'état interne du processeur ciblé ou le paramétrage du banc d'injection. En réponse, il est nécessaire de concevoir des méthodes d'analyse qui intègrent les conditions de réalisation de l'effet des fautes.

Problématique de réalisation des campagnes d'injection

Les chemins d'attaque qui sont déterminés par l'analyse indiquent seulement quelles instructions (ou autres éléments, selon le niveau d'abstraction) doivent être fautées. Ces indications ne sont pas suffisantes pour réaliser une attaque physique et il est nécessaire de traduire les chemins d'attaque en paramètres physiques, en particulier en ce qui concerne les délais d'injection de fautes. Cette étape est chronophage et souvent laissée aux experts. De plus, la synchronisation entre le banc d'injection et la cible de l'attaque peut être encore plus difficile pour une attaque multi-faute, car les fautes peuvent également altérer le temps d'exécution des programmes. Ainsi, il est crucial de disposer de techniques pour automatiser la calibration du banc d'injection, la réalisation des campagnes, puis traitement des résultats des tentatives d'attaques.

Approche automatique pour faciliter la réalisation de campagnes d'injection

En réponse à ces deux problématiques, cette thèse étudie la capacité d'une approche qui fait le lien entre l'analyse de programme et la mise en œuvre des chemins d'attaque identifiés. Nous présentons notre approche appelée SAMPLAI dans le chapitre 4. SAMPLAI intègre une extension de notre analyse capable de prendre en compte, pendant la recherche du chemin d'attaque, des caractéristiques plus précises du modèle de faute et ses conditions particulières pour la réalisation d'un saut d'instruction au niveau ISA. À l'issue de l'analyse, des chemins d'attaque plus réalistes peuvent être déterminés, mais

qui nécessitent d'être traduits en paramètres d'injection. À ce titre, SAMPLAI propose une méthode de conversion des chemins en ensembles de paramètres temporels d'injection qui repose sur l'instrumentation de code directement sur la carte cible. Les campagnes de fautes sont ensuite automatisées avec une méthode comprenant les communications avec la plateforme d'injection et la carte cible, les injections des fautes et le traitement des résultats.

2.4 Conclusion

De nombreuses techniques d'analyse ont été proposées pour évaluer la sécurité des systèmes face aux attaques par injection de fautes. Selon la technique utilisée, des compromis apparaissent nécessairement, que ce soit sur le nombre de chemins d'exécution fautés testés, la taille du code ou du design analysé, ou encore le nombre de fautes considérées. Les approches basées sur la simulation sont intrinsèquement limitées quant au nombre d'exécutions fautées qu'elles peuvent tester, ce qui empêche une couverture exhaustive. Dans le contexte des attaques multi-fautes, il devient impossible de tester toutes les combinaisons de fautes, nécessitant ainsi une sélection stratégique des positions de fautes les plus pertinentes [12], [13]. Les simulations peuvent également être utiles pour valider des attaques théoriques identifiées par d'autres approches. Les techniques utilisant l'exécution symbolique, en revanche, permettent de couvrir plus simplement l'ensemble des exécutions fautées à l'aide de formules. Cependant, selon la méthode utilisée, le nombre de mutants [101] ou de bifurcations [14] explosent avec le nombre de fautes considérées pendant l'analyse. Une solution comme FASE [15], [94] qui utilise un encodage des fautes, permet de passer à l'échelle, mais nécessite des optimisations pour réduire la complexité des formules et reste limitée à dix instructions fautées au maximum. Enfin, les méthodes basées sur la vérification de modèles sont également limitées par la complexité computationnelle qui augmente avec la taille du modèle et le nombre de fautes [17]. En conséquence, ces approches sont souvent contraintes en utilisant des modèles bornés, rendant difficile l'analyse de sécurité pour des attaques comprenant de nombreuses fautes.

En réponse aux limitations des méthodes actuelles et en tenant compte des récentes avancées dans les plateformes d'injection de fautes, cette thèse propose de nouvelles approches pour évaluer la sécurité dans des scénarios d'attaques comportant de nombreuses fautes. Deux contributions sont présentées. La première est une méthode d'analyse au niveau ISA, appelée SAMVA, qui se concentre sur l'identification de chemins d'attaque im-

pliquant de multiples fautes ayant pour effets des sauts d'instructions [19]. Cette approche est basée exclusivement sur de l'analyse statique, une méthode d'analyse peu étudiée dans le domaine des attaques en fautes. La seconde contribution, nommé SAMPLAI, est une technique qui automatise la recherche de paramètres d'injection et la réalisation physique des attaques multi-fautes. En s'appuyant sur les chemins d'attaque théoriques identifiés par SAMVA, SAMPLAI convertit les fautes en un ensemble de paramètres temporels en s'adaptant aux spécificités de la cible, facilitant ainsi la réalisation concrète des attaques.

Chapitre 3

Analyse statique pour la détermination de chemins d'attaque multi-fautes

Les travaux récents autour des plateformes d'injection de fautes ont montré la possibilité d'injecter plusieurs fautes à des moments distincts au cours d'une même attaque [4] et de corrompre plusieurs instructions consécutives [5]-[10]. En conséquence, de nombreuses contre-mesures utilisant le principe de redondance des opérations [2], [3] peuvent être contournées, ouvrant la voie à des attaques plus complexes. Les techniques d'évaluation de la sécurité des programmes face aux attaques multi-fautes restent toutefois limitées par le nombre de fautes considérées ou la taille du code analysé.

Pour répondre à cette problématique, nous proposons SAMVA, une méthode d'analyse statique permettant d'identifier des chemins d'attaque dans des scénarios multi-fautes. Les chemins d'attaque identifiés permettent de réaliser des attaques d'accessibilité, où le flot de contrôle est détourné pour atteindre des sections du programme afin de réaliser des actions interdites initialement. SAMVA est capable de modéliser des scénarios où un attaquant est capable d'injecter plusieurs fautes ayant pour effet le saut d'instructions. Contrairement aux autres approches présentées dans la section 2.2, notre analyse repose exclusivement sur de l'analyse statique, avec pour objectif de passer à l'échelle même lorsque les fautes sont nombreuses. Nous proposons également une implémentation de SAMVA basée sur le projet *angr* [87] et en utilisant les langages Python (8,5k lignes) et C++ (1,2k lignes).

Notre approche opère au niveau de l'ISA et prend en charge l'architecture ARMv7-M. SAMVA analyse le binaire à évaluer selon les capacités de l'attaquant et de ses objectifs. Les objectifs sont exprimés sous forme d'une liste d'adresses d'instructions qui doivent impérativement être atteintes, ainsi qu'un ensemble d'adresses à ne pas exécuter, représentant les mécanismes de détection d'attaques. La largeur d'une faute désigne le nombre

d'instructions sautées par une unique injection, et les capacités de l'attaquant sont définies par la largeur minimale et maximale d'une faute, ainsi que par le délai minimum possible entre deux injections successives. Ces paramètres dépendent des capacités du banc d'injection utilisé. L'analyse statique s'appuie sur des heuristiques de recherche de chemins dans un graphe orienté qui représente le programme et les effets potentiels des fautes. Les chemins candidats identifiés sont ensuite analysés pour déterminer une configuration de fautes suffisante, définie par les positions et les largeurs des fautes à injecter, afin de rendre le chemin d'attaque réalisable. Le résultat final de l'analyse est un ensemble de chemins d'attaque répondant aux contraintes d'injection spécifiées par l'utilisateur.

Pour valider ces chemins d'attaque, nous utilisons un simulateur de fautes basé sur `gem5` [102]. Nous évaluons SAMVA sur huit versions d'un programme de vérification de code PIN issues de la suite de benchmarks FISSC [20] contenant diverses contre-mesures logicielles, en testant de nombreuses capacités d'attaquants. Nous montrons que SAMVA est capable, dans un temps très réduit, de trouver dans toutes les implémentations, même les plus sécurisées, des configurations de fautes à injecter pour atteindre un objectif. Ces travaux ont été publiés dans la workshop international COSADE en 2023 [19].

Le modèle de menace considéré est détaillé dans la section 3.1. La section 3.2 explique le fonctionnement de notre méthode d'analyse pour l'identification des chemins d'attaque. La section 3.3 présente notre dispositif expérimental ainsi que les résultats obtenus. Enfin, nous discutons des travaux apparentés dans la section 3.4.

3.1 Modèle de menace

Cette section présente plus en détail le modèle de faute de saut d'instructions qui est utilisé dans l'analyse SAMVA. Nous discutons également des effets potentiels de ce type de faute sur le flot de contrôle, afin de mieux comprendre la menace que représentent de telles attaques sur la sécurité d'un programme.

3.1.1 Définition du saut d'instruction

Les effets des fautes peuvent être modélisés à différents niveaux (niveau logique, RTL, code assembleur, code source) en utilisant une approche ascendante : comprendre l'impact des fautes en se basant sur les effets observables et la connaissance du code testé. De nombreux travaux de recherche se sont concentrés spécifiquement sur la modélisation au

niveau ISA [41], [53]-[56].

À ce niveau, les fautes peuvent entraîner divers effets sur le programme, tels que le remplacement d'une instruction par une autre ou, plus fréquemment, le saut d'instruction [38], [41], [77]. Ce modèle suppose que certaines instructions soient « sautées », comme si leur exécution n'avait pas eu lieu, ou qu'elles aient été remplacées par des instructions équivalentes à un *NOP* (pour *no operation*), qui n'ont aucun effet.

Les articles susmentionnés rapportent les effets rencontrés lors d'une attaque comportant seulement une faute, mais des travaux récents montrent qu'un banc d'injection de fautes moderne peut mener à la corruption de plusieurs instructions consécutives. Les impulsions électromagnétiques peuvent entraîner le saut de plusieurs instructions consécutives, de deux instructions jusqu'à une douzaine [5], [6], [8], [103]. Les techniques d'injection de fautes par laser peuvent également conduire au saut de quelques instructions choisies [9] ou d'un grand nombre d'instructions consécutives, de 1 à 300 en fonction de la durée de l'impulsion laser [7]. Des sauts d'instructions multiples, allant de quelques instructions choisies à près d'une centaine, peuvent également être réalisés en utilisant des moyens d'injection moins coûteux tels que l'altération du signal de l'horloge [10].

Le modèle de faute du saut d'instruction est une abstraction au niveau ISA de différentes manifestations en pratique. Ces manifestations incluent, par exemple, le remplacement d'une instruction par une autre sans effet sur le contexte d'exécution, le rejeu d'instructions, ou encore la substitution du registre de destination par un registre inutilisé. Notre analyse opère au niveau ISA et s'abstrait des problématiques liées à la réalisation concrète des attaques. Le chapitre 4 présente une méthodologie permettant de conduire des attaques sur du matériel à partir des vulnérabilités identifiées.

3.1.2 Effets des sauts d'instructions sur le flot de contrôle

Le modèle du saut d'instruction est puissant, car il offre de nombreuses façons de corrompre le flot de contrôle d'un programme durant son exécution. Lorsqu'il est possible de réaliser plusieurs sauts d'instructions, un adversaire peut combiner les effets de ces fautes afin de réaliser des attaques encore plus puissantes : Péneau et al. [78] démontrent que si des sauts d'instructions de tailles variables sont injectés avec suffisamment de précision et en grand nombre, un programme binaire peut être compromis de multiples façons. Les auteurs prouvent également que la programmation orientée *NOP* est complète au sens de Turing.

Afin de montrer des conséquences possibles des sauts d'instructions sur le flot de

contrôle d'un programme, nous prenons quelques exemples de codes assembleur accompagnés de leur graphe de flot de contrôle (CFG pour *control flow graph* en anglais). Un CFG modélise le flot de contrôle entre les blocs de bases (BB) d'un programme. Un BB est défini comme une séquence de code linéaire de taille maximale ne contenant aucun branchement et un unique point d'entrée au début de la séquence. Un CFG est un graphe orienté, $G = (N, E)$, où chaque nœud $n \in N$ correspond à un BB et chaque arc $e = (n_i, n_j) \in E$ correspond à un possible transfert de contrôle entre deux BB [104]. Le CFG est une abstraction contenant un sur-ensemble des chemins d'exécution possibles d'un programme et peut être utilisé pour modéliser les effets des fautes impactant le flot de contrôle initial.

```
1  # BB1
2  ...
3  mov r0, 123
4  bl <BB3>     # Appel de fonction
5  # BB2
6  ldr r3, [sp, 4]
7  ...
8  # BB3
9  add r0, 456
10 bx lr        # Retour de fonction
```

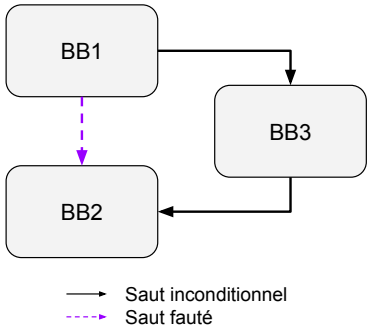


FIGURE 3.1 – Saut inconditionnel fauté

Au niveau binaire, un programme comporte en général deux types d'instructions de saut : les branchements inconditionnels et les branchements conditionnels réalisant un saut en fonction de l'état du programme. Pour ces derniers, si la condition de saut n'est pas satisfaite, alors l'exécution continue avec les instructions suivantes. La figure 3.1 représente un extrait de code assembleur Arm et son CFG associé. Ce code contient trois BB et lors d'une exécution nominale, l'exécution suit le chemin *BB1* - *BB3* - *BB2*. Néanmoins, un saut d'instruction sur le branchement de la ligne 4 force l'exécution à continuer en séquence et à exécuter le BB2. Le chemin est donc *BB1* - *BB2*, ce qui est impossible sans faute. Ce flot de contrôle illustre la capacité d'une injection de fautes à détourner le flot de contrôle afin d'ignorer des appels de fonction ou contourner les réactions après une vérification de sécurité.

Concernant les sauts conditionnels, il y a deux branchements possibles et les fautes permettent de forcer un des branchements. La figure 3.2 montre un autre extrait de code

```
1  # BB1
2  ...
3  ldrsb.w r3, [r3]
4  cmp     r3, #0
5  ble.n <BB3>
6  # BB2
7  movs    r2, 4
8  b <BB4>
9  # BB3
10 movs    r3, 5
11 # BB4
12 ...
```

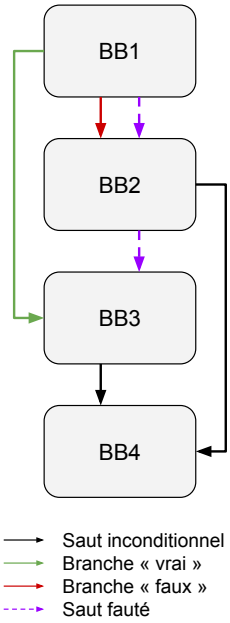


FIGURE 3.2 – Saut conditionnel fauté

et son CFG associé. Dans cet exemple, il y a quatre BB représentant une structure de type *if-then-else* d'un langage de plus haut niveau. Le BB1 représente l'évaluation de la condition du saut puis termine avec l'instruction de branchement (ligne 5). Sauter cette instruction revient à forcer le branchement « faux », désignant le chemin pris lorsque la condition n'est pas satisfaite *BB1* - *BB2*. En revanche pour forcer la branche « vrai », c'est-à-dire le chemin *BB1* - *BB3*, il faut réussir à satisfaire la condition du saut. À noter que selon l'agencement du code, il est possible d'exécuter les deux blocs désignés par le branchement conditionnel en sautant également l'instruction de saut inconditionnel à la fin du premier bloc (ici *BB2*, donc les lignes 5 et 8).

Le saut d'instruction permet de manipuler le flot de contrôle à l'intérieur d'une procédure, mais aussi entre les procédures et tirer profit de l'agencement du binaire. De manière générale, lors d'un appel d'une fonction, l'adresse de retour est stockée dans un registre spécifique comme le registre de lien ou dans la pile. Pour optimiser la taille du programme binaire, les fonctions sont agencées les unes à la suite des autres donc sauter l'instruction de retour de fonction, qui permet retourner à la fonction appelante, peut permettre de continuer l'exécution vers la fonction placée à la suite. La figure 3.3 montre un code assembleur et son CFG où les nœuds sont les fonctions complètes et les arcs sont les appels et leurs retours. Dans cet exemple, le retour de fonction est effectué en

```
1  # Dernier bloc de la fonction F1
2  ...
3  mov     sp, r7
4  pop {r7, pc}    # Retour de la
       fonction et restauration de la
       pile
5
6  # Premier bloc de la fonction F2
7  push {r7, lr}   # Sauvegarde des
       registres sur la pile
8  sub     sp, 24
9  ...
```

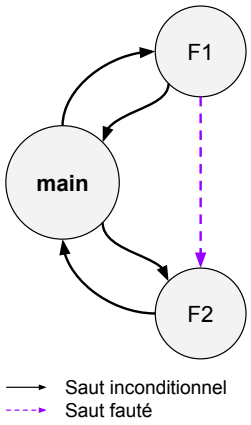


FIGURE 3.3 – Détournement du flot de contrôle entre deux fonctions

réécrivant le pointeur d'instruction (le registre *pc*) à partir des données stockées dans la pile. En sautant l'instruction qui effectue le retour de la fonction *F1* vers le *main* (ligne 4), l'exécution peut se poursuivre dans la fonction suivante *F2*. Toutefois, pour assurer une exécution cohérente sans plantage, l'état du processeur (pile et registres), doit rester cohérent. Par conséquent, sauter une instruction qui effectue la restauration de la pile ou la sauvegarde des registres peut entraîner un état incohérent, pouvant provoquer un plantage.

3.2 Méthode statique pour déterminer des chemins d'attaque

Dans cette section, nous présentons d'abord une vue d'ensemble de l'approche mise en œuvre dans SAMVA. Ensuite, nous expliquons la technique de modélisation des effets potentiels des fautes sur le flot de contrôle. Nous terminons par expliquer l'algorithme permettant de déterminer les positions des fautes à injecter lors de l'attaque.

3.2.1 Vue d'ensemble

La figure 3.4 donne une d'ensemble de l'analyse dédiée à l'identification de chemins d'attaque. L'analyse prend en entrée le programme binaire, les objectifs de l'attaquant et sa capacité à injecter des fautes. Le résultat de SAMVA est une liste contenant au maximum `N` chemins d'attaque, `N` étant aussi défini par l'utilisateur. Un chemin d'attaque

contient la position des fautes avec leurs largeurs correspondantes (notée ci-après *fw* pour *fault width* en anglais) à injecter lors l'exécution du programme sur la cible pour atteindre les objectifs de l'attaquant.

Les objectifs de l'attaquant, désignés comme les *spécifications de l'exploit* dans la figure 3.4, se compose des éléments suivants : (1) une liste ordonnée de BB devant être atteints au cours de l'exécution et dans l'ordre spécifié ; (2) un ensemble de BB ne devant jamais être exécutés, car pouvant correspondre à des codes de détection d'attaque en fautes par exemple.

Dans ces travaux, nous considérons un attaquant capable d'injecter des fautes multiples et précises, menant au saut d'une ou de plusieurs instructions consécutives. La distance entre deux fautes injectées, ainsi que le nombre minimal et maximal d'instructions sautées par injection dépendent de la méthode d'injection utilisée. La capacité de l'attaquant à injecter des fautes est exprimée à l'aide de trois paramètres (cf. les paramètres des fautes dans la figure 3.4) : (1) `fw_min` indique le nombre minimal d'instructions sautées pour une injection de faute ; (2) `fw_max` indique le nombre maximal d'instructions sautées ; (3) `f_min_dist` exprime le nombre minimal d'instructions devant être exécutées entre deux injections de fautes, pouvant être imposé par le moyen d'injection utilisé. Par exemple, la configuration de Dutertre et al. [7] (cf. section 3.1) serait spécifiée en affectant `fw_min=1` et `fw_max=300`. La `f_min_dist` serait définie en fonction de la fréquence du processeur ciblé et du temps de rechargement du banc d'injection de fautes.

SAMVA est conçu spécifiquement pour trouver des chemins d'attaque avec des fautes qui ont pour effet de détourner le flot de contrôle. Tout d'abord, un CFG est construit à partir du programme au format binaire et sert de support pour la suite. Ce CFG est étendu et ses arcs sont annotés afin de refléter les éventuels effets des injections de fautes. Ce graphe étendu est noté ci-après ECFG (pour *extended* CFG). Ensuite, des chemins d'attaque candidats sont calculés à partir de l'ECFG et des objectifs de l'attaquant. Enfin, l'analyse cherche un ensemble de chemins d'attaque finaux qui respecte les contraintes relatives aux capacités de l'attaquant. Chaque chemin d'attaque donné en sortie prend la forme d'une liste d'instructions qui reflète le chemin d'exécution fauté, avec la position et la largeur des fautes à injecter.

3.2.2 Modélisation des effets des fautes

Notre approche implémentée dans SAMVA commence par la création du CFG à partir du programme binaire. Le CFG caractérise tous les chemins d'exécution possibles en

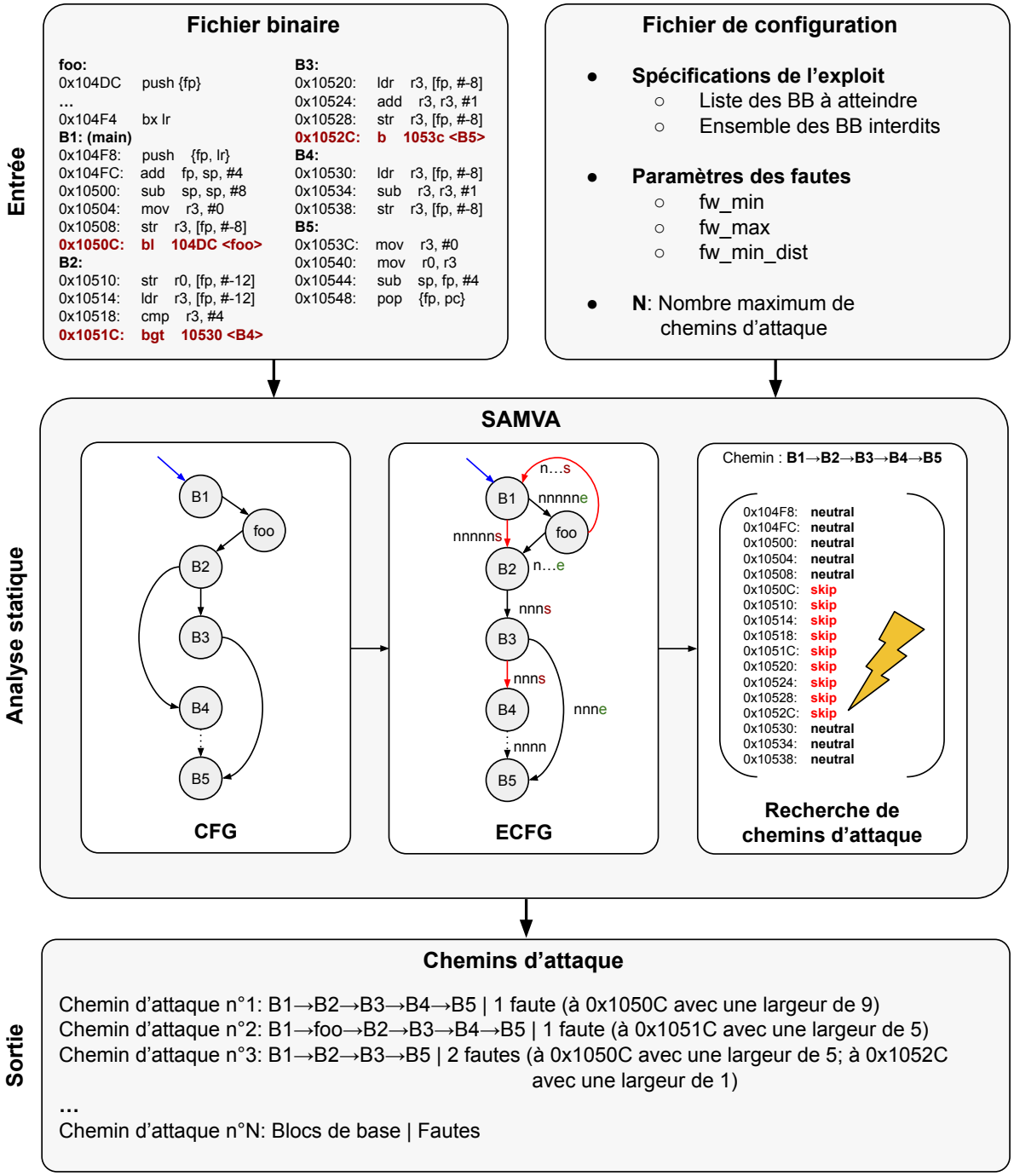


FIGURE 3.4 – Vue d'ensemble de la plateforme. Dans l'exemple de code, les BB ciblés sont [B1, B2, B5] et l'ensemble des BB interdits est vide. Quelques chemins d'attaque trouvés sont donnés pour illustrer le format de sortie.

l'absence d'attaque. Nous l'appelons CFG nominal dans la suite. Il peut être obtenu par une analyse purement statique ou par une combinaison d'analyse statique et symbolique. Dans SAMVA, nous utilisons l'infrastructure logicielle *angr* [87] pour le construire.

Comme montré avec les exemples de la section 3.1, la possibilité de sauter l'exécution d'instructions bien choisies permet à un attaquant de modifier le flot de contrôle d'un programme de manière à forcer un chemin d'exécution existant, ou d'en créer un nouveau. Nous modélisons ces effets potentiels des sauts d'instructions en générant le graphe étendu (ECFG) à partir du CFG nominal. Cette étape est indépendante de l'attaque, car elle modélise tous les effets potentiels des fautes sans tenir compte de la capacité de l'attaquant ou des objectifs. Dans ce qui suit, nous détaillons les deux transformations effectuées sur le CFG nominal pour générer l'ECFG qui est ensuite utilisé par notre heuristique de recherche de chemins d'attaque.

Modélisation des détournements de flot de contrôle

La capacité de pouvoir sauter l'exécution d'instructions de branchement permet à un attaquant de forcer l'exécution des instructions qui se trouvent en mémoire juste après ces instructions de branchement. Lorsqu'un saut inconditionnel est la cible d'une faute, l'exécution se poursuit dans le bloc placé après en mémoire. Pour refléter cet effet, un arc est ajouté au graphe entre ces deux blocs. Ce flot de contrôle permis par l'injection d'une faute est illustré dans la figure 3.4 avec l'insertion d'un nouvel arc entre *B1* et *B2* dans l'ECFG. De la même manière, lorsqu'un saut d'un branchement conditionnel est réalisé, l'exécution continue avec l'instruction qui suit immédiatement le branchement, correspondant au cas où la condition n'est pas satisfaite. SAMVA ne reposant actuellement sur aucune analyse de flot de données, la direction prise par les branchements conditionnels doit être déterminée statiquement pour identifier les chemins réalisables. Afin de maitriser totalement le chemin d'exécution suivi, les branches prises sont considérées systématiquement sautées et les arcs correspondants sont supprimés de l'ECFG. Par exemple, dans l'ECFG illustré à la figure 3.4, l'arc reliant *B2* à *B4* a été supprimé. Cependant, il reste possible, en utilisant plusieurs sauts d'instructions, d'atteindre la cible du branchement conditionnel si celle-ci est située à une adresse plus grande. Dans l'ECFG de la figure 3.4, un attaquant devrait sauter le branchement à la fin de *B2* ainsi que *B3* pour atteindre *B4* à partir de *B2*.

Une limitation de notre approche, qui repose exclusivement sur l'analyse du flot de contrôle et le saut d'instruction, est son incapacité à gérer les branchements conditionnels

arrière, c'est-à-dire forcer l'exécution du BB cible d'un branchement conditionnel lorsque celui-ci se situe à une adresse inférieure dans l'agencement de la mémoire. Pour répondre à ce problème, une analyse du flot de données serait nécessaire afin de déterminer s'il est possible de manipuler la condition en utilisant uniquement des fautes de saut d'instruction, ou si une faute en dehors de notre modèle, comme l'inversion de la condition de branchement, est requise.

Annotations des arcs

Cette étape consiste à annoter les arcs de l'ECFG pour indiquer si le flot de contrôle correspondant résulte ou non d'un saut de l'instruction de branchement du BB source de l'arc. Pour définir l'annotation des arcs, nous considérons les types d'instructions suivants :

- *execute* (`e`) est le type des instructions qui doivent être exécutées ;
- *skip* (`s`) est le type des instructions dont l'exécution doit être sautée ;
- *neutral* (`n`) est le type des instructions qui peuvent être soit fautées soit exécutées sans affecter le flot de contrôle à la fin de leur BB.

Chaque instruction d'un bloc de base est typée afin d'automatiser l'identification des positions des fautes tout en respectant les capacités de l'attaquant définies par l'utilisateur. Le type *neutral* permet de considérer davantage de solutions possibles. Les arcs entre les BB sont annotés en associant un type à chaque instruction du BB source, dans leur ordre d'exécution. Les instructions de branchement sont systématiquement typées comme *skip* ou *execute*, reflétant ainsi si l'instruction de branchement est exécuté ou non. Par défaut, toutes les autres instructions d'un bloc de base sont typées *neutral*. Ces annotations sont également illustrées sur la figure 3.4.

Bien que l'annotation par défaut soit suffisante pour rechercher des chemins d'attaque réalisables, nous affinons la stratégie de typage pour éviter les sources de plantage à l'exécution d'une attaque. Notamment, des mises à jour incohérentes du pointeur de pile pendant l'exécution entraîne souvent un plantage du programme attaqué. Aussi, une adresse de retour incohérente peut faire dévier l'exécution du chemin d'exécution prévu. En conséquence, nous définissons deux règles optionnelles de typage :

- *`R1` : Exécution des mises à jour du pointeur de pile.* Les instructions qui écrivent dans le registre du pointeur de pile (registre `SP`), telles que les instructions `push` et `pop`, sont toujours typées comme *execute*. Cela garantit que la mémoire allouée à la pile est ensuite désallouée. Pour les architectures ARM, le retour d'une fonction

appelée vers son appelante utilise l'instruction unique `pop pc` ou équivalente. Cette instruction permet de récupérer l'adresse de retour sur la pile, mettre à jour le pointeur de pile et de retourner à la fonction appelante. L'exécution des instructions `pop` étant forcée par cette règle de typage, cela garantit qu'un appel de fonction est soit ignoré, soit que le retour à l'appelant sera correctement exécuté si un plantage n'intervient pas avant.

— *`R2` : Exécution des retours de fonction à l'aide du registre de lien (ou* link register*).* Dans le cas d'une fonction « feuille », c'est-à-dire une fonction qui n'appelle pas de fonction, le retour à la fonction appelante est souvent réalisé à partir du registre de lien, si la pression sur les registres le permet. Ce registre `lr`, défini par l'instruction d'appel `bl`, peut être directement utilisé en utilisant une instruction `bx lr` ou équivalente. Cela se produit lorsque le registre de lien `lr` n'est pas sauvegardé sur la pile en raison d'une faible pression sur les registres. Cette règle supplémentaire type toutes les instructions `bx lr` comme *execute*. Par conséquent, une fonction feuille qui ne sauvegarde pas le registre de lien sur la pile est soit ignorée, soit le retour à la fonction appelante est correctement exécuté.

Ces deux règles sont implémentées sous la forme de module dans SAMVA, pouvant être activées ou non lors de l'analyse.

3.2.3 Recherche de chemins d'attaque

Pour des raisons de performance, nous n'explorons pas de façon exhaustive l'ensemble des chemins pour identifier des vulnérabilités. À la place, nous utilisons une heuristique permettant d'explorer l'ECFG construit précédemment. Nous commençons par générer un ensemble de chemins candidats qui sont conformes aux objectifs de l'attaque. Les annotations des arcs présentes sur chaque chemin sont ensuite utilisées pour déterminer la position et la largeur des injections de fautes à effectuer. La position et la largeur des injections de fautes doivent, d'une part, être cohérentes avec le type des instructions du chemin, et d'autre part, avec les capacités de l'attaquant.

Génération de chemins candidats

Un ensemble de chemins candidats est généré par l'exploration de l'ECFG. Ces chemins doivent atteindre, dans le bon ordre, les BB spécifiés dans l'objectif de l'attaquant et éviter les BB interdits. Idéalement, les chemins candidats doivent permettre de positionner

facilement l'injection des fautes en espaçant au maximum les instructions typées *execute* et *skip*, et réduire le nombre d'injections de fautes en favorisant les instructions *neutre* et *execute*. Par conséquent, nous associons à chacun des arcs de l'ECFG un poids en fonction de son annotation.

— Poids = 1 : toutes les instructions sont typées *neutral* ;
— Poids = 2 : les instructions sont seulement typées *neutral* ou *execute* ;
— Poids = 3 : les instructions sont uniquement typées *neutral* ou *skip* ;
— Poids = 4 : certaines instructions sont typées *skip* et d'autres *execute*.

Cette politique de pondération des arcs oriente la recherche de chemins vers des chemins d'attaque dont le positionnement des fautes est plus simple. Ainsi, pour chaque paire de blocs de base successifs issus de la liste des blocs de base ciblés, un ensemble temporaire de chemins est récupéré en utilisant l'algorithme des plus courts chemins [105] en fonction des poids des arcs. La complexité de la recherche des K premiers chemins les plus courts dans un CFG contenant N_{BB} blocs de base est alors de $\mathcal{O}(K{N_{BB}}^3)$. Lors de l'analyse

L'ensemble final de chemins candidats complets $P_{candidate\_paths}$ passant par tous les blocs de base spécifiés dans l'objectif de l'attaquant est alors construit en faisant le produit cartésien des ensembles temporaires. Nous combinons itérativement les ensembles correspondant à des blocs de base consécutifs et retenons les K chemins ayant les poids les plus faibles. Cet ensemble final est composé de chemins candidats qui ne garantissent pas nécessairement la possibilité de positionner les injections de fautes requises par rapport aux capacités de l'attaquant. L'étape suivante vise à trouver un ensemble valide d'injections de fautes à réaliser pour faire d'un chemin candidat un chemin d'attaque.

Positionnement des injections de fautes

La détermination des fautes à injecter pour rendre réalisable un chemin candidat donné est basée sur les types des instructions présentes sur les annotations des arcs et sur les capacités de l'attaquant. Pour un chemin candidat donné, nous construisons ce que nous appellons une « trace d'exécution » qui est une liste de paires $\langle adresse,\ type \rangle$. Le positionnement des injections de fautes vise à trouver la position et la largeur des fautes à injecter de telle sorte que les instructions typées *skip* soient couvertes par une faute et que les instructions typées *execute* soient en dehors de toute faute. Les instructions typées *neutre* peuvent être couvertes ou non par une faute, sans impacter le flot de contrôle.

La largeur de toute faute injectée doit être comprise dans [`fw_min`, `fw_max`]. Par conséquent, il existe potentiellement de nombreuses possibilités pour la position et la largeur

des injections des fautes, comme le montre la figure 3.5. Néanmoins, la distance entre deux fautes consécutives doit être supérieure ou égale à la distance minimale `f_min_dist`. Pour des raisons de performance, il n'est pas réaliste de calculer l'ensemble des positions et largeurs de fautes possibles. En conséquence, nous utilisons une approche en deux étapes pour déterminer un ensemble de solutions de taille fixe : i) d'abord, nous utilisons des règles simples pour filtrer rapidement les chemins qui n'aboutiront pas à un résultat, sur la base de la distance entre les instructions typées *execute* et *skip* ; ii) l'ensemble des configurations de fautes (c'est-à-dire la position et la largeur) est exploré à l'aide d'un algorithme de retour en arrière (ou *backtracking* en anglais) afin de trouver une configuration valide qui rende faisable un chemin candidat.

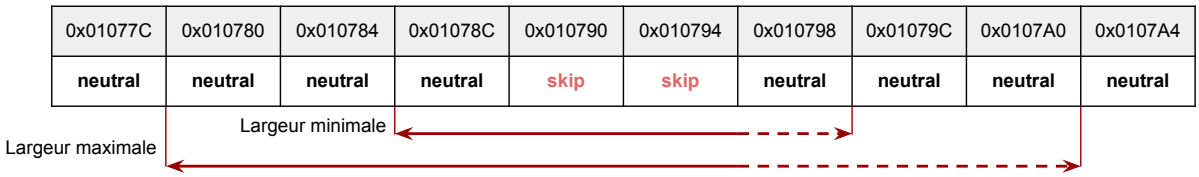


FIGURE 3.5 – Exemple de positionnement des fautes sur une trace avec $\texttt{fw\_min} = 3, \texttt{fw\_max} = 5$

Vérification de l'insolubilité. Nous utilisons ces règles simples pour détecter rapidement l'insolubilité du problème de positionnement des fautes sur une trace d'exécution :

— S'il y a au moins une instruction typée *skip* entre deux instructions i_0 et i_1 typées *execute*, alors la distance entre i_0 et i_1 doit être supérieure ou égale à `fw_min` la largeur minimale d'une faute. Sinon, toute faute couvrant l'instruction typée *skip* a au moins un impact sur i_0 ou i_1, donc le problème de positionnement des fautes est insoluble ;

— S'il existe au moins une instruction de type *execute* entre deux instructions i_0 et i_1 typées *skip*, alors la distance entre i_0 et i_1 doit être supérieure ou égale à `f_min_dist` la distance minimale entre deux fautes. Dans le cas contraire, le problème de positionnement des fautes est insoluble.

Algorithme de retour en arrière (backtracking). L'algorithme essaie de placer des fautes afin de couvrir toutes les instructions typées *skip* dans une trace. Ainsi, nous cherchons à construire une solution, consistant en une liste de fautes ayant chacune une position et une largeur. De plus, ces fautes doivent respecter les contraintes de taille et d'espacement des fautes.

Algorithme 1 Algorithme de positionnement des fautes utilisant le retour en arrière

```
1:  function FAULT_POSITIONING(trace, f_candidates, f_params)
2:      next_pos ← find_next_skip(trace, f_candidates)
3:      if next_pos = ∅ then
4:          return True
5:      for fw ∈ range(f_params.fw_max, f_params.fw_min, -1) do
6:          for pos ∈ range(next_pos, next_pos - fw, -1) do
7:              fault ← <pos, fw>
8:              if is_valid(fault, trace, f_candidates, f_params) then
9:                  f_candidates.push(fault)
10:                 if fault_positioning(trace, f_candidates, fault_params) then
11:                     return True
12:                 else
13:                     candidate_faults.pop()
14:     return False
```

Une solution est construite de manière incrémentale avec une approche basée sur le retour en arrière en utilisant la récursivité. L'algorithme 1 donne un aperçu de notre implémentation. Tout d'abord, la position de la prochaine instruction typée *skip* qui n'est pas encore couverte par une faute est retrouvée à partir de la trace d'exécution (ligne 2). Ensuite, nous faisons varier la largeur (ligne 5) et la position de la faute (ligne 6) afin de trouver une configuration de faute valide.

Pour déterminer si la configuration de faute courante est valide, les propriétés suivantes sont vérifiées (ligne 8) :

— la position de la faute doit être contenue dans la trace, c'est-à-dire prendre en compte les limites de la trace ;
— la faute ne doit pas couvrir une instruction typée *execute* ;
— la faute ne doit pas se superposer à la faute précédente (s'il y en a une) et leur distance doit être supérieure à la distance minimale entre deux fautes.

Les appels récursifs s'arrêtent lorsqu'une solution respectant toutes les contraintes et couvrant toutes les instructions typées *skip* est obtenue. Cela se produit lorsque la fonction `find_next_skip` ne trouve plus aucune instruction non couverte typée *skip* (lignes 4 et 10). Au cours de ce processus, si nous découvrons que la solution actuelle ne sera pas valide, nous revenons en arrière, ce qui se traduit par le retour à l'étape précédente en supprimant la dernière faute validée (ligne 13) et en essayant une autre configuration de fautes à la place. Les algorithmes de retour en arrière utilisent la méthode de recherche

en profondeur. Afin de minimiser le nombre de fautes nécessaires pour réaliser l'attaque, nous explorons d'abord, comme il est possible de constater dans l'ordre des boucles, les positions possibles en partant de celle de l'instruction à couvrir, puis nous faisons varier sa largeur, en commençant par la plus large.

Dans un souci de performance, nous procédons à quelques optimisations afin de réduire l'espace des configurations de fautes possibles. Premièrement, lors de la validation de la position d'une faute, nous vérifions également s'il existe une instruction typée *skip* plus loin dans la trace à une distance inférieure à `fw_min_dist` qui engendrerait une violation des contraintes. Par conséquent, même si la configuration de la faute est valide avec les fautes déjà choisies, elle est rejetée pour éviter des appels récursifs inutiles. Deuxièmement, nous décomposons notre trace d'exécution en plusieurs sous-traces, plus faciles à analyser, qui sont traitées indépendamment. Nous appliquons une coupe dans la trace d'exécution lorsque (1) deux instructions typées *skip* ne sont séparées que par des instructions typées *neutral*, et (2) la distance entre ces deux instructions est supérieure à deux fois la largeur maximale de la faute plus la distance minimale entre deux fautes afin d'éviter toutes interactions.

Rétrécissement de la largeur des fautes. Notre algorithme de positionnement des fautes tente de rendre les fautes aussi larges que possible afin d'en réduire le nombre. Il peut donc trouver des solutions valides qui couvrent néanmoins inutilement des instructions typées *neutral*. Pour cette raison, nous appliquons une dernière passe qui diminue la largeur des fautes lorsque cela est possible. Elle déplace le début et la fin d'une faute vers la première et la dernière instruction typée *skip* tout en respectant la contrainte de la largeur minimale de la faute (`fw_min`). On obtient ainsi des fautes plus petites, potentiellement plus faciles à réaliser et qui réduisent le risque de sauter des instructions menant à un plantage.

3.2.4 Implémentation de SAMVA

Nous avons implémenté notre méthode dans un outil développé en Python, comportant 8,5k lignes de code. Cet outil est basé sur l'infrastructure *angr* [87], qui permet de décompiler le binaire cible et de construire un CFG. *angr* fournit également plusieurs représentations intermédiaires des instructions et des BB, permettant d'extraire des informations sur la nature des instructions et des opérations sur les registres. Le CFG est modélisé à l'aide de la bibliothèque *NetworkX* [106], qui intègre de nombreux algorithmes

et opérations sur le graphe. Lors de la génération du ECFG, *NetworkX* permet d'annoter les arcs du graphe avec des objets Python qui inclus le typage des instructions. Pour des raisons de performance, l'algorithme de positionnement des fautes est écrit en C++ sous forme d'un module CPython. Ce module, composé de 1,2k lignes de code C++, s'intègre au reste de l'outil écrit en Python. L'outil est utilisable en ligne de commande et prend en entrée le binaire cible, un fichier de spécification d'attaques au format JSON, ainsi que les capacités de l'attaquant. Les chemins d'attaque résultats sont ensuite enregistrés sous forme de fichiers CSV.

3.3 Expérimentations

Dans cette section, nous évaluons l'efficacité de SAMVA et analysons la résistance de contre-mesures aux attaques multi-fautes. Nous commençons par décrire le dispositif expérimental, en détaillant les applications ciblées, les capacités de l'attaquant considérées, ainsi que notre méthode d'évaluation. Ensuite, nous présentons les résultats obtenus.

3.3.1 Dispositif expérimental

Nous présentons les programmes que nous ciblons pour ces expérimentations, les paramètres d'injection utilisés ainsi que la méthodologie pour analyser les résultats.

Programmes cibles

Nous évaluons notre analyse sur les programmes de vérification de code PIN issus du projet FISCC [20]. Cette collection de programmes contient huit implémentations de `VerifyPIN`, une implémentation non protégée (nommée V0), comme illustrée dans la figure 3.6 et sept autres implémentations intégrant différentes contre-mesures (nommées V1 à V7). Les programmes de vérification du code PIN comparent le code PIN fourni par l'utilisateur `g_userPin` et le code PIN de la carte `g_cardPin` à l'aide de la fonction `byteArrayCompare` (ligne 3). La variable `g_authenticated`, initialement à 0 est mise à 1 en cas de succès. Le nombre de tentatives est contrôlé par la variable `g_ptc`, initialement fixée à 3 et décrémentée après chaque tentative d'authentification échouée. L'authentification n'est plus autorisée si `g_ptc` atteint zéro afin d'éviter les attaques par force brute sur le code PIN. Pour les implémentations protégées, c'est-à-dire les versions supérieures à V0, une routine spéciale est appelée lorsqu'une attaque est détectée par une contre-mesure.

Cette routine attribue la valeur « vrai » à une variable ajoutée à toute version protégée et nommée `g_countermeasure`. Grâce à cette variable, l'évaluateur est en mesure de savoir si une attaque a été détectée par une des contre-mesures. Les contre-mesures mises en œuvre sont décrites ci-dessous. Le tableau de la figure 3.7 indique les contre-mesures intégrées dans chaque version de `VerifyPIN` ainsi que le nombre d'instructions, de blocs de base et d'arcs du CFG initial et le nombre d'arcs ajoutés par l'ECFG pris en compte dans l'analyse au niveau binaire.

```
1  g_authenticated = 0;
2  if(g_ptc > 0) {
3      if(byteArrayCompare(g_userPin, g_cardPin)) {
4          g_ptc = 3;
5          g_authenticated = 1;
6      }
7      else {
8          g_ptc--;
9      }
10 }
```

FIGURE 3.6 – Code source de la fonction `VerifyPIN` sans contre-mesure (V0)

Les contre-mesures employées par les programmes cibles sont les suivantes :

— Booléens renforcés (**HB** pour *Hardened Booleans*) : Les booléens sont encodés avec deux constantes dont la distance de Hamming est élevée afin de rendre plus difficile le passage d'une valeur à une autre avec une unique injection. Cette protection vise les fautes modifiant les données.

— Suivi d'exécution (**SC** pour *Step counter*) : certaines variables servant à compter des étapes de l'exécution sont ajoutées au code afin de se protéger contre les attaques perturbant l'intégrité du flot de contrôle. Le nombre d'itérations de la boucle est vérifié à la sortie de la boucle dans les versions `V2` à `V5`. Dans la version `V7`, toutes les instructions et tous les flots de contrôle sont protégés par de telles variables.

— Expansion des appels (**IC** pour *Inlined Calls*) : les appels de fonction sont *inlinés*, ce qui consiste à remplacer un appel de fonction par le code de cette fonction. Cela permet de se prémunir contre un saut d'instruction sur l'instruction d'appel d'une fonction. Cela peut réduire la surface d'attaque, car il n'y a plus les instructions servant au passage des paramètres lors des appels.

— Duplication de variable sensible (**BC** pour *Backup copy*) : le nombre de tentatives restantes est dupliqué afin d'empêcher les attaques avec une faute unique visant spécifiquement le compteur de tentatives.
— Duplication des tests et appels (**DT** pour *Double Test*) : l'appel à la fonction de vérification des codes PIN et tous les tests sont dupliqués pour éviter qu'une faute unique détourne le flot de contrôle.

L'objectif de l'attaquant est d'obtenir une authentification sans connaître le code PIN de l'utilisateur et sans déclencher de contre-mesures. En conséquence, notre analyse recherche les fautes permettant de détourner le flot de contrôle du programme afin d'exécuter le code d'authentification (lignes 4 et 5) sans jamais exécuter les mécanismes de détection d'attaque. Pour les expérimentations, nous identifions manuellement les blocs de base ciblés et interdits pour chaque implémentation de `VerifyPIN`. L'exploit commence au début de la fonction de vérification et la liste des BB ciblés contient le BB affectant la variable `g_authenticated` à vraie dans la fonction `VerifyPIN` et le BB qui suit l'appel de la fonction `VerifyPIN` dans la fonction `main` afin de garantir le bon retour à la fonction appelante. L'ensemble des BB interdits comprend la routine liée à la détection des fautes qui modifie la variable `g_countermeasure`.

Les huit versions de `VerifyPIN` sont compilées pour jeu d'instructions Arm Thumb (ARMv7-M). Le cross-compilateur utilisé est GNU GCC gnueabi en version 8.5.0. Nous désactivons toutes les optimisations du compilateur (`-O0`) afin d'éviter l'altération des contre-mesures logicielles, ainsi que l'utilisation d'instructions conditionnées qui ne sont pas encore prises en charge par SAMVA.

	HB	**SC**	**IC**	**DT**	**BC**	Nbr. d'instr	Nbr. de BB	Nbr. d'arcs
V0						142	24	46 (+12)
V1	✓					162	30	57 (+15)
V2	✓	✓				172	32	58 (+15)
V3	✓	✓	✓			158	30	54 (+13)
V4	✓	✓	✓	✓		221	41	79 (+20)
V5	✓	✓			✓	241	47	87 (+22)
V6	✓		✓		✓	177	36	68 (+17)
V7	✓	✓	✓		✓	306	66	140 (+38)

FIGURE 3.7 – Description de la suite `VerifyPIN` avec les contre-mesures incluses, leur nombre d'instructions, les BB et les arcs ECFG (+ arcs ajoutés au CFG original) au niveau binaire

Paramètres d'injection de fautes

Pour chaque implémentation de `VerifyPIN`, nous considérons de nombreux paramètres d'injection de fautes correspondant à différentes capacités d'attaquant. Nous faisons varier la largeur des fautes possibles (à l'aide des paramètres de faute `fw_min` et `fw_max`) ainsi que la distance minimale entre deux fautes consécutives (à l'aide du paramètre `fw_min_dist`). Notre objectif est d'observer la sensibilité des contre-mesures utilisées par rapport aux paramètres d'injection de fautes nécessaires pour effectuer une attaque.

Soit W l'ensemble des valeurs possibles de la largeur de la faute mesurée en nombre d'instructions. Il est défini de la manière suivante : $W := \{1\} \bigcup \{2n : n \in \mathbb{N}^* \mid n \leq 32\}$. Ainsi, la largeur minimale vaut soit 1, soit une valeur paire entre 2 et 64 ; la largeur maximale varie en fonction de la largeur minimale et de tous les nombres pairs entre 2 et 64 également, de sorte que : $\{(\texttt{fw\_min}, \texttt{fw\_max}) \in W \times W \mid \texttt{fw\_min} \leq \texttt{fw\_max}\}$. Enfin, la distance minimale entre deux injections de fautes, mesurée également en nombre d'instructions, vaut une puissance de 2, de telle sorte que : $\texttt{fw\_min\_dist} \in \{2^n : n \in \mathbb{N} \mid n \leq 5\}$.

En outre, nous exécutons SAMVA sur toutes les versions de `VerifyPIN` en considérant trois stratégies de typage des instructions (cf. section 3.2.2) : la première, dénotée `défaut`, ne contenant aucune règle de typage supplémentaire ; la deuxième comprend la règle `R1` qui force l'exécution des mises à jour du pointeur de pile ; la troisième, dénotée `R1 + R2`, applique à la fois les règles `R1` et `R2`. En résumé, nous testons SAMVA sur un total de 3366 paramètres de fautes distincts, sur chacun des huit fichiers binaires, pour chacune des trois stratégies de typage d'instructions, avec le rétrécissement de largeur des fautes activé ou désactivé.

Méthodologie d'évaluation

Nous itérons à travers les possibles paramètres de fautes, les stratégies de typage et les binaires comme décrit précédemment. Pour un couple composé d'un binaire et d'un paramètre de faute, SAMVA essaie de générer un ensemble de `N` chemins d'attaque distincts. Dans nos expériences, `N` est égal à 30, ce qui signifie que nous nous attendons à obtenir jusqu'à 30 chemins d'attaque, en fonction des possibilités offertes par les instructions utilisées et le placement du code des binaires. La méthode de classification des chemins résultats est illustrée dans la figure 3.8.

L'ensemble des chemins d'attaque renvoyé par l'analyse peut être vide si notre ana-

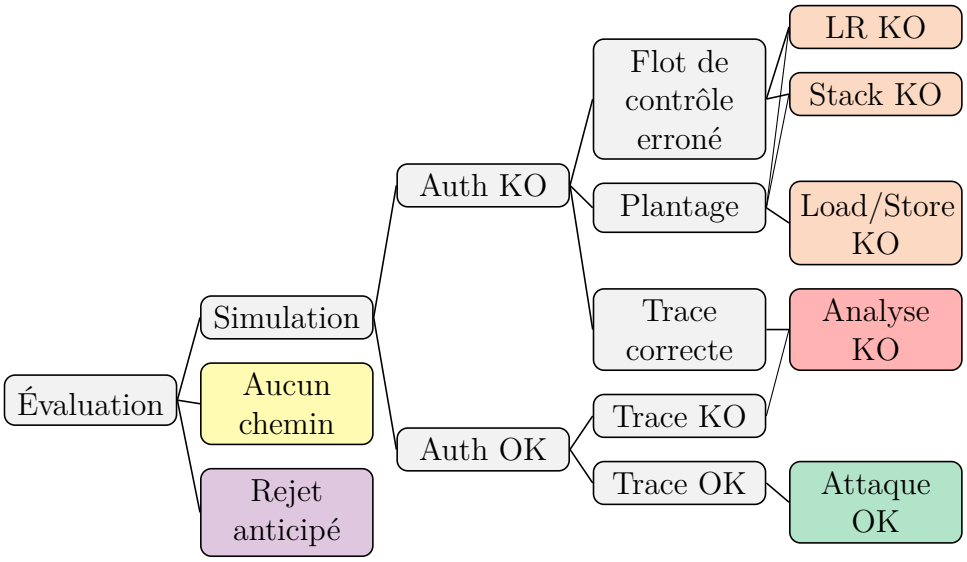


FIGURE 3.8 – Arbre de décision pour la classification des résultats d'attaques

lyse ne trouve aucun chemin d'attaque pour les paramètres d'analyse (classe « Aucun chemin » □). Aussi, les paramètres de faute peuvent être inconsistants pour le code binaire considéré. Cela arrive quand `fw_min`, la largeur minimale de la faute, est supérieure au nombre d'instructions séparant le point de départ et la première instruction cible de l'attaque (classe « Rejet anticipé » □), puisque l'exécution des BB ciblés est obligatoire.

Si l'ensemble résultat n'est pas vide, les chemins d'attaque sont validés par une simulation du modèle de faute, en l'occurrence le saut d'instruction. Le simulateur utilisé dans nos expériences est une version modifiée de `gem5` [102], capable de sauter l'exécution d'instructions choisies dans un ordre d'occurrence spécifié. Il prend en entrée le programme binaire analysé et les fautes qui doivent être injectées pour réaliser un chemin d'attaque. Lorsque que la simulation se termine sans signaler de plantage, nous examinons les valeurs des variables d'état du programme `VerifyPIN`. Si `g_authenticated` vaut « vrai » et `g_countermeasure` vaut « faux », alors on considère que l'authentification a été accordée (nœud « Auth OK »). Dans ce cas, la trace d'exécution résultant de la simulation et celle prévue par l'analyse sont comparées afin de s'assurer qu'elles correspondent (nœud « Attaque OK » □). Nous arrêtons d'itérer sur l'ensemble résultat après une première attaque simulée réussie. Dans le cas d'un plantage lors de la simulation ou si l'authentification a échoué (nœud « Auth KO »), la trace d'exécution obtenue par la simulation et celle retournée par l'analyse sont également comparées. Lorsqu'elles ne correspondent pas, nous déterminons la raison du plantage ou de la divergence du flot de contrôle en inspectant la

dernière instruction en commun entre les deux traces, exécutée par la simulation. Selon l'instruction et l'état du système, nous catégorisons l'erreur en trois : le nœud « LR KO » pour un retour de fonction erroné qui utilise le registre de lien, le nœud « Stack KO » lors d'un plantage lié à un état de pile incohérent et le nœud « Load/Store KO » lors d'un plantage causé par une lecture ou une écriture illégale. Dans ces trois cas, le résultat est de couleur orange (les nœuds « Plantage » et « Flot de contrôle erroné »). Enfin, nous mesurons également les cas d'échec de notre analyse pour deux cas particuliers (nœud « Analyse KO »). Le premier cas est celui où l'authentification échoue malgré la correspondance entre les traces d'exécution et l'authentification attendue par l'analyse. Le second cas est celui où l'authentification réussit, mais où les traces ne correspondent pas. Ces cas ne sont que des vérifications (*sanity check*), qui ne devraient pas se produire. Nous ne les avons pas rencontrés lors de nos expérimentations.

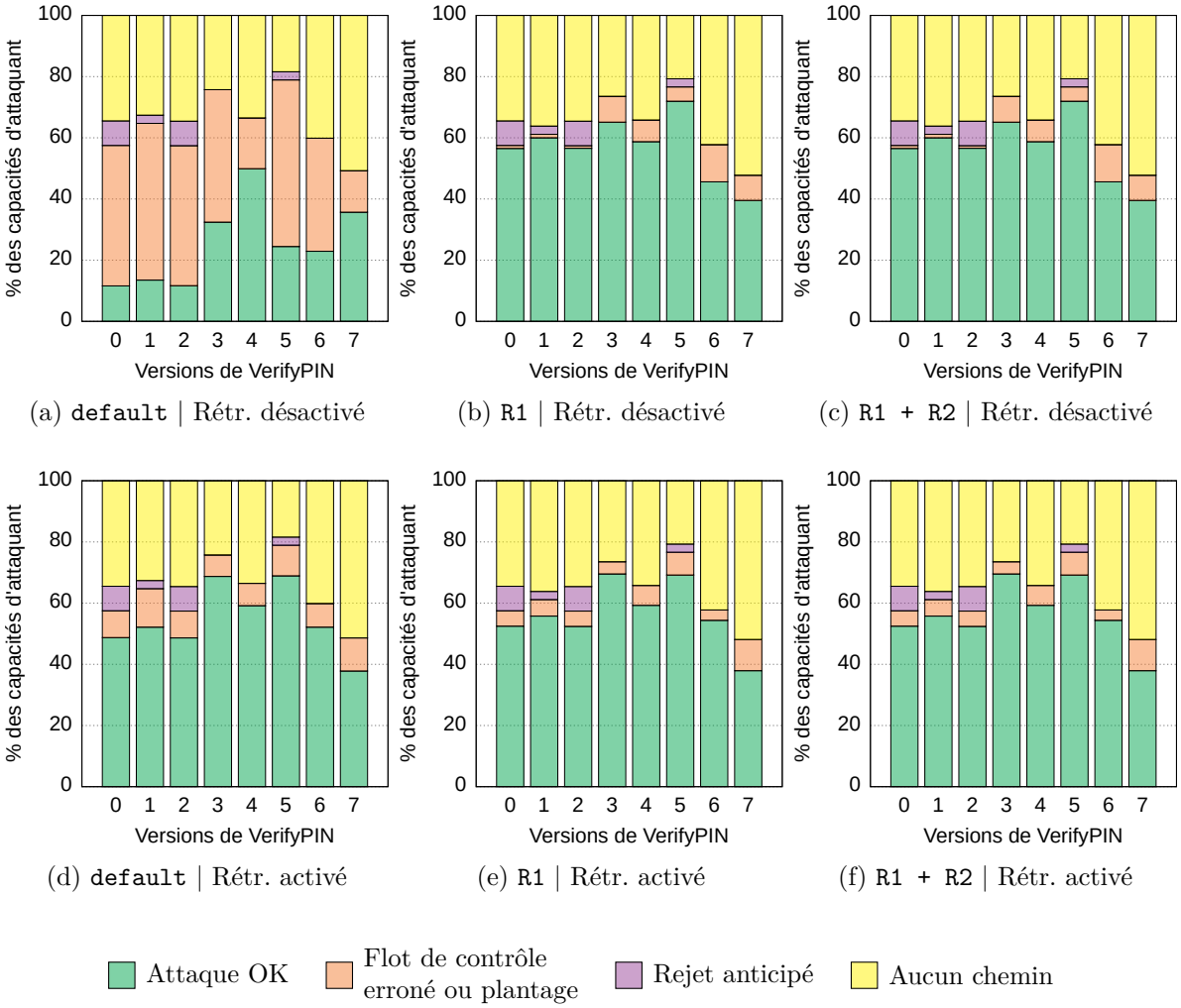
3.3.2 Résultats expérimentaux

Les résultats expérimentaux incluent une présentation de la répartition des paramètres d'injection conduisant à des chemins d'attaque réussis. Nous examinons ensuite la forme des fautes observées, avant de terminer par une mesure du temps d'analyse nécessaire à SAMVA pour identifier ces chemins d'attaque.

Résultats de l'évaluation des chemins d'attaque

Les expérimentations menées visent à mesurer la capacité de SAMVA dans la recherche de chemins d'attaque dans les différents benchmarks en fonction des différentes stratégies de typage d'instructions. Nous avons utilisé deux spécifications d'attaque : la première vise uniquement à exécuter l'assignation de la variable d'authentification (`g_authenticated`) à `true`, tandis que la seconde, qui étend la première, cherche à exécuter l'authentification et la réinitialisation de la variable du compteur d'essais (`g_ptc`) à sa valeur initiale de 3. En plus de s'authentifier, la deuxième spécification permet de rendre l'attaque totalement invisible sur le système. Pour chacune des attaques, nous considérons les trois stratégies avec et sans rétrécissement des fautes.

La figure 3.9 montre les résultats de l'évaluation obtenus avec la première stratégie selon la classification présentée dans la figure 3.8. Les trois histogrammes du haut représentent les résultats des trois stratégies sans rétrécissement des fautes et ceux du bas avec rétrécissement.

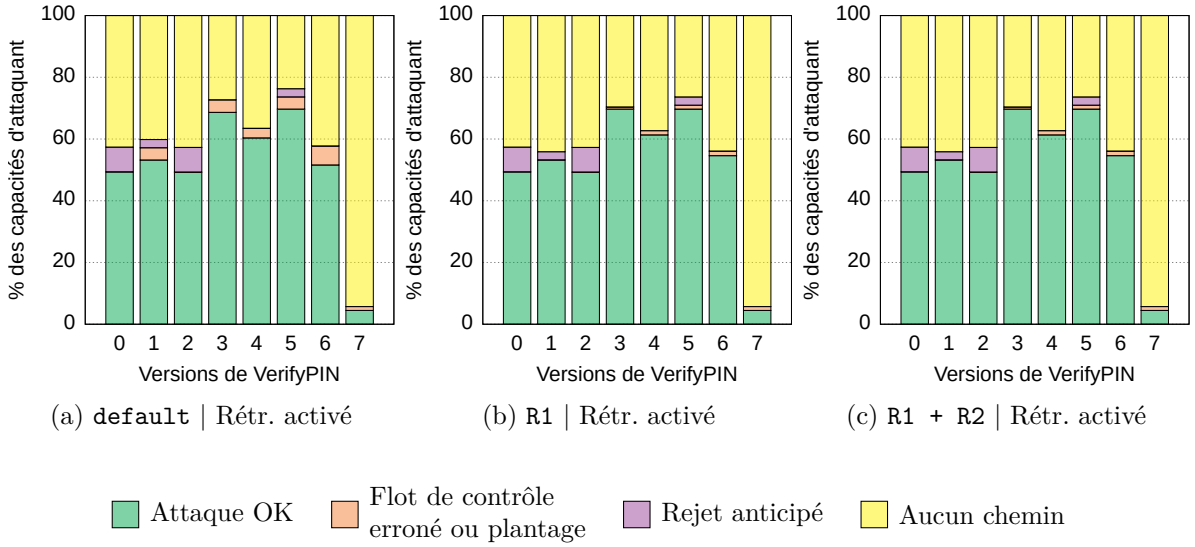


(a) `default` | Rétr. désactivé (b) `R1` | Rétr. désactivé (c) `R1 + R2` | Rétr. désactivé

(d) `default` | Rétr. activé (e) `R1` | Rétr. activé (f) `R1 + R2` | Rétr. activé

FIGURE 3.9 – Classification des résultats des recherches de chemin d'attaque. L'attaque vise l'affectation de `g_authenticated` à « vrai ».

D'abord, nous constatons que nous sommes en mesure de trouver des chemins d'attaque avec toutes les stratégies et configurations d'analyse. La principale différence entre la stratégie par défaut et les autres stratégies préservant l'exécution des mises à jour des pointeurs de pile est le nombre plus élevé de plantage au cours de la simulation utilisant la stratégie par défaut. Cependant, le rétrécissement des fautes semble atténuer sensiblement le nombre de plantages en réduisant le nombre d'instructions devant être sautées. Cela montre bien que le rétrécissement des fautes réduit le risque de sauter une instruction nécessaire à la bonne exécution du programme, telle que l'allocation de mémoire pour la pile. Néanmoins, nous observons que cette technique a un impact uniquement lorsque aucune



(a) `default` | Rétr. activé (b) `R1` | Rétr. activé (c) `R1 + R2` | Rétr. activé

FIGURE 3.10 – Classification des résultats des recherches de chemin d'attaque. L'attaque vise l'affectation de `g_authenticated` à « vrai » et l'affectation de `g_ptc` à 3.

règle de typage d'instructions n'est activée. Dans le cas contraire, les stratégies garantissent l'exécution de certaines instructions, ce qui réduit le nombre de plantages. Enfin, la stratégie `R1 + R2` affiche un nombre de chemins d'attaque identifiés similaire à la stratégie `R1`. Cela s'explique par l'agencement du code binaire, où la fonction de vérification qui compare l'entrée de l'utilisateur avec le code PIN du système (`byteArrayCompare`) est positionnée avant la fonction `VerifyPIN` et la fonction `main`. De fait, SAMVA n'a pas besoin de faire de saut entre les procédures et donc, le retour de fonction était déjà normalement exécuté. Les plantages restants sont uniquement dus à des accès invalides à la mémoire en raison des sauts d'instructions qui peuvent altérer le calcul des adresses à accéder. Pour charger ou stocker une valeur, l'adresse est généralement stockée dans un registre défini au préalable par une ou plusieurs instructions. En effet, si l'une de ces instructions est sautée, un accès illégal peut survenir et provoquer un plantage.

La figure 3.10 montre la classification des résultats obtenus avec la deuxième spécification d'attaque avec le rétrécissement des fautes activé. Nous pouvons observer que moins de chemins d'attaque ont été identifiées, notamment pour la version `V7`. Après la compilation, les codes relatifs à l'authentification et l'affectation du compteur d'essai partagent le même BB, à l'exception de la version `V7`, où chacune de ces affectations sont dans des BB différents. Les chemins d'attaque pour la version `V7` doivent sauter quelques instructions entre l'exécution des deux BB. Ceci limite les paramètres d'injection pouvant capables de

positionner une faute suffisamment petite pour réaliser l'attaque.

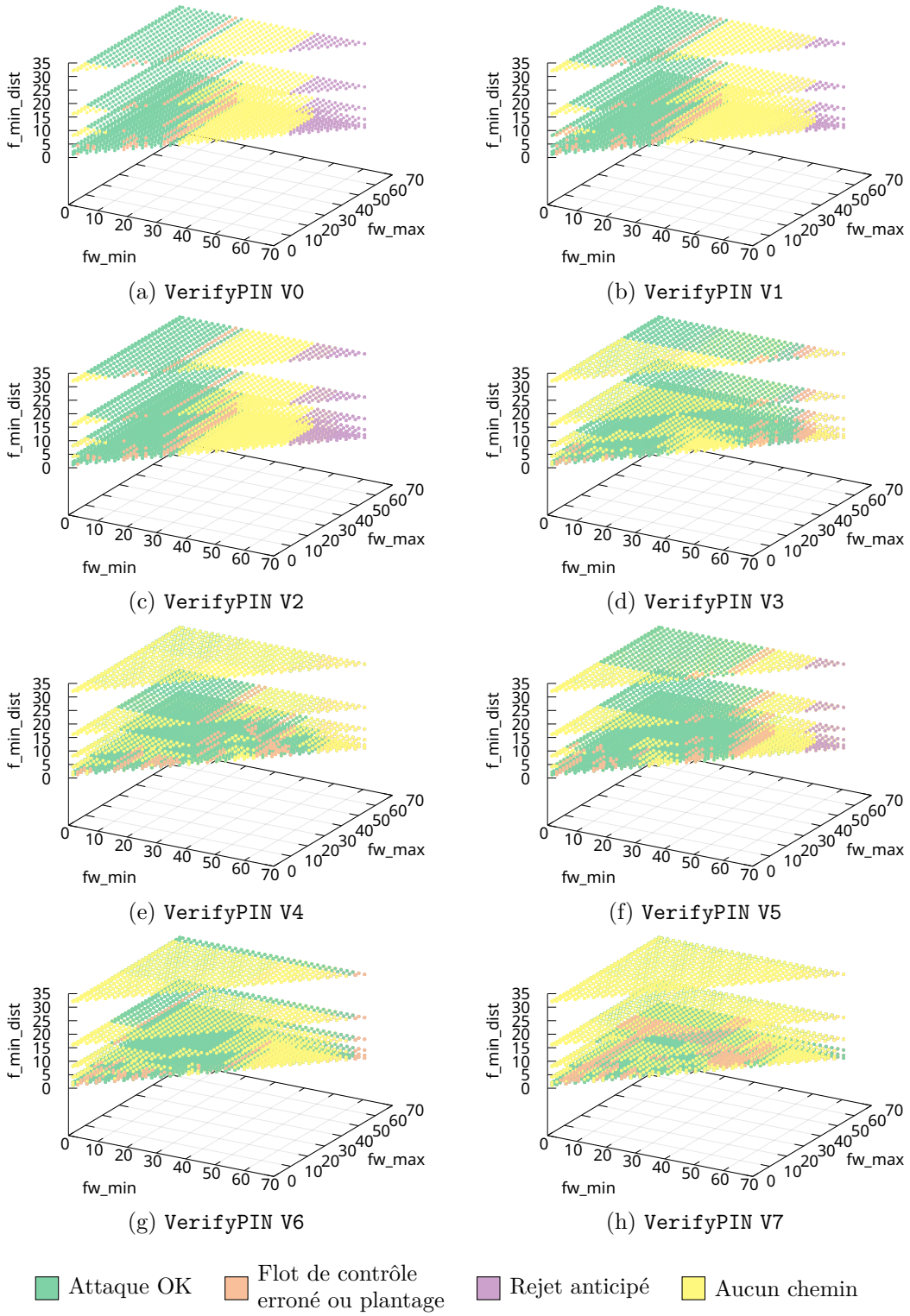
Dans la suite de cette section, nous nous concentrons sur les résultats obtenus pour la première spécification d'attaque, où l'authentification est réussie sans déclencher de contre-mesures, sans considération pour le compteur d'essais. Plus précisément, nous utilisons la stratégie qui permet de trouver le plus grand nombre de chemins d'attaque réussis, c'est-à-dire celle qui comporte la règle de typage supplémentaire R1 + R2 accompagnée du rétrécissement des fautes. Les résultats de cette évaluation sont illustrés dans la figure 3.9f. Au travers de ces résultats, nous étudions l'impact des paramètres de fautes, la forme des fautes issues des chemins ainsi que les nombres de fautes requis pas ces chemins.

Étude des paramètres d'injection de fautes

La figure 3.11 est une représentation alternative des résultats pour les huit binaires, qui montre la classification des résultats de l'analyse en fonction des trois paramètres de faute considérés (`fw_min`, `fw_max`, `fw_min_dist`). D'après la répartition des points verts, le schéma général que nous pouvons observer est que plus `fw_min` et `fw_min_dist` sont petits, plus `fw_max` est grand, plus il y a de possibilités de trouver des chemins d'attaque qui aboutissent à des attaques réussies. Ceci s'explique par la flexibilité offerte par ces paramètres dans la taille et le positionnement des fautes. On constate que les versions `V1`, `V2` et `V5` ont des résultats similaires à `V0`, ce qui signifie que les contre-mesures mises en œuvre n'ont qu'un effet limité contre les sauts multiples. Nous pouvons également constater que dans `V4` et `V6`, la distance entre deux fautes devient le facteur principal pour la faisabilité de l'attaque. Cela peut s'expliquer par la nécessité de faire plusieurs fautes si `fw_max` ne permet pas de faire une faute suffisamment grande. Enfin, pour la `V7`, nous trouvons moins de configurations de fautes qui conduisent à des attaques réussies. La contre-mesure d'intégrité du flot de contrôle à grain fin incluse dans cette version oblige à sauter plusieurs petits ensembles d'instructions et réduit donc la surface d'attaque.

Caractéristiques des configurations de faute réussies

Nous étudions les caractéristiques des configurations de fautes qui conduisent à des attaques réussies. La figure 3.13 présente le nombre de fautes nécessaires pour chaque attaque réussie, rangé par ordre croissant afin de montrer le nombre minimum de fautes. Nos résultats montrent que les versions `V0` à `V3`, `V5` et `V6` peuvent être attaquées avec une seule faute. Les versions `V4` et `V7` peuvent être attaquées avec au moins deux fautes. Le typage des instructions obtenu pour un chemin d'attaque donné est responsable du



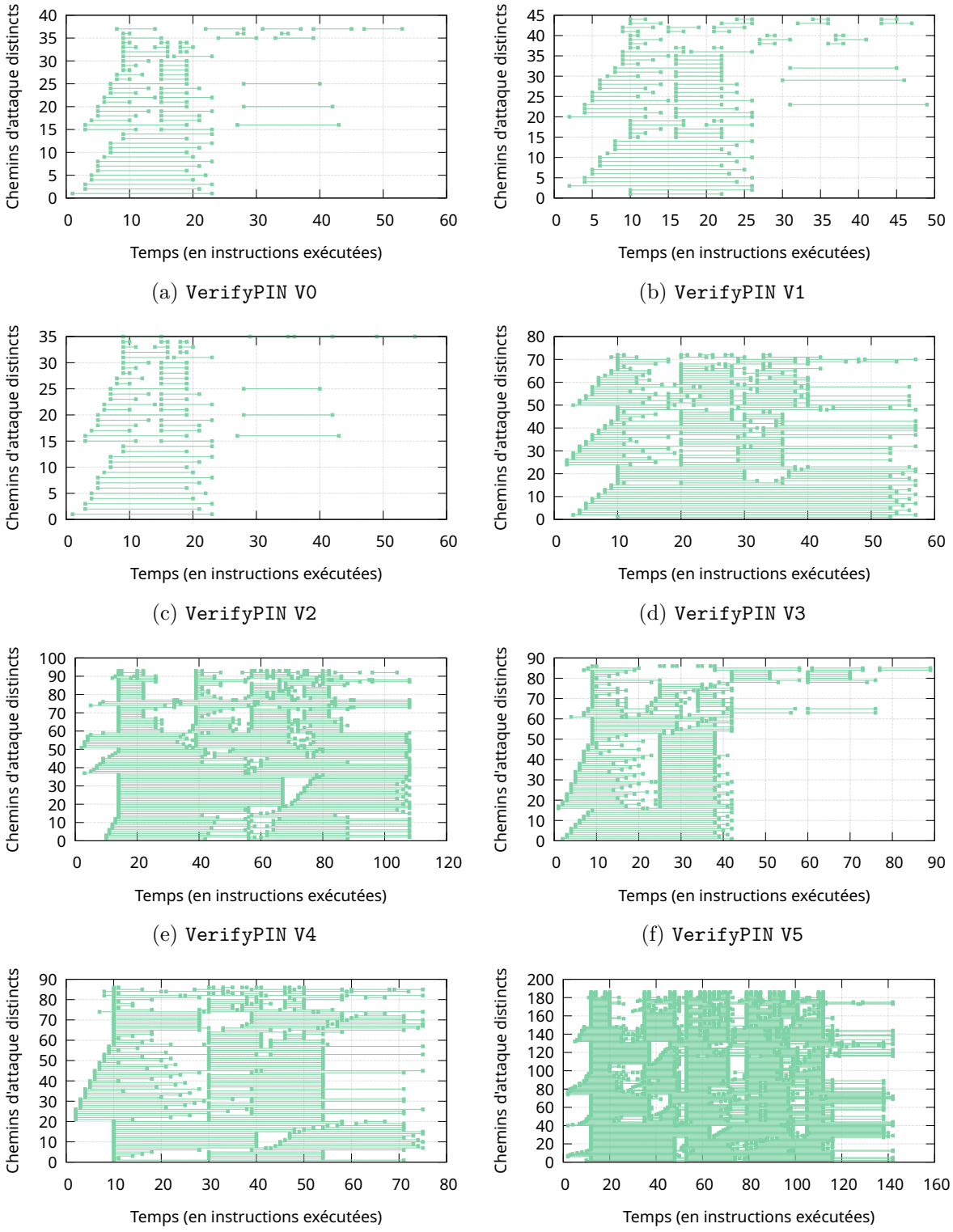
(a) `VerifyPIN V0` (b) `VerifyPIN V1`

(c) `VerifyPIN V2` (d) `VerifyPIN V3`

(e) `VerifyPIN V4` (f) `VerifyPIN V5`

(g) `VerifyPIN V6` (h) `VerifyPIN V7`

FIGURE 3.11 – Résultats de chaque configuration testée, par version de `VerifyPIN`, en utilisant la stratégie `R1 + R2` et en activant le rétrécissement des fautes



(a) `VerifyPIN V0`

(b) `VerifyPIN V1`

(c) `VerifyPIN V2`

(d) `VerifyPIN V3`

(e) `VerifyPIN V4`

(f) `VerifyPIN V5`

(g) `VerifyPIN V6`

(h) `VerifyPIN V7`

FIGURE 3.12 – Attaques uniques identifiées pour chaque version de `VerifyPIN`

nombre minimum de fautes. Par exemple, si deux instructions sont typées *skip* avec une instruction typée *execute* entre les deux, alors deux fautes sont forcément nécessaires. En fonction de l'agencement du code binaire et des instructions supplémentaires induites par les contre-mesures, un chemin d'attaque peut contenir de telles contraintes, ce qui se traduit par un nombre de fautes requis plus élevé pour les versions `V4` et `V7`.

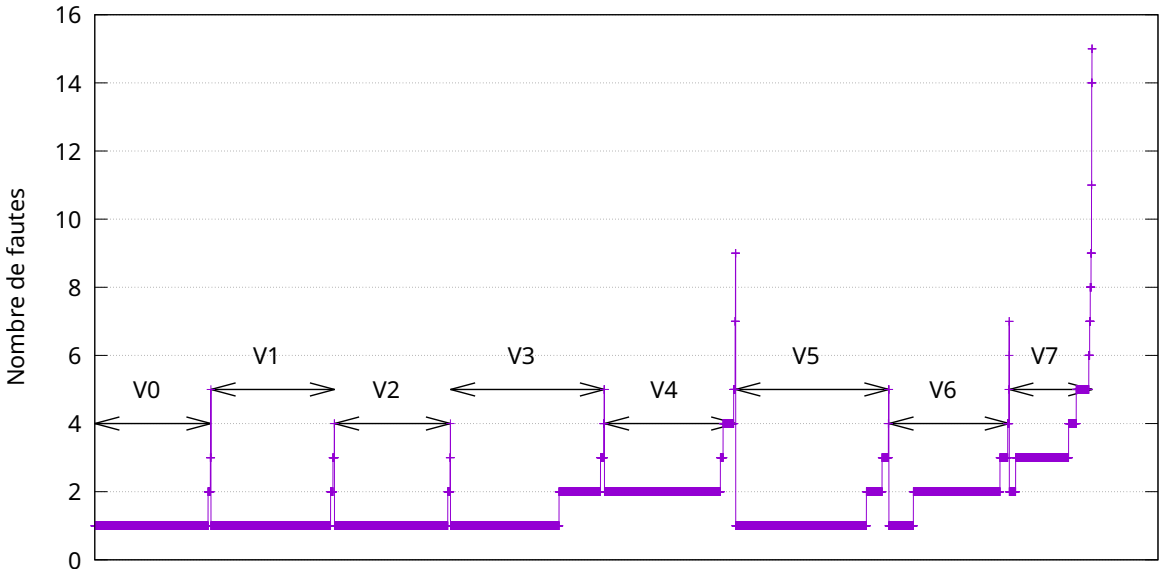


FIGURE 3.13 – Nombre de fautes nécessaires pour chaque attaque réussie trouvée pour chaque version de `VerifyPIN`, rangé par ordre croissant, en utilisant la stratégie `R1 + R2` avec le rétrécissement des fautes activé

Pour mieux comprendre les effets du typage des instructions sur les positionnements des fautes, la figure 3.12 représente les caractéristiques des fautes sur les chemins d'attaque validés. Chaque attaque est représentée horizontalement. L'axe des abscisses représente le temps (plus précisément, les instructions exécutées consécutivement). Chaque segment désigne une faute dont la largeur correspond aux nombre d'instructions sautées. Par exemple, la version `V1` peut être attaquée avec une seule faute de largeur 12 (segment du bas, allant de x=10 à x=22) ; mais aussi avec quatre fautes plus étroites représentées à y=41 : une faute allant de x=10 à x=15 suivie de trois fautes de largeur 3 aux temps x=23, x=33, et x=43. Les attaques sont triées verticalement en fonction du nombre de fautes et par ordre croissant.

Ces différents chemins d'attaque peuvent correspondre au même flot de contrôle, bien que nous puissions voir certains motifs. En prenant la version `V0` comme exemple, nous pouvons remarquer qu'en fonction des paramètres d'injection de fautes, SAMVA peut

choisir de faire une longue faute pour couvrir toutes les instructions typées *skip* ou de faire plusieurs fautes plus petites pour les couvrir individuellement. Il en résulte des points de passage obligatoires dans le flot de contrôle, qui se manifestent graphiquement sous la forme d'une colonne dans les figures, parce qu'aucune faute n'est autorisée à couvrir cette section du chemin d'attaque. Enfin, nous considérons uniquement les chemins entièrement prévisibles dans notre analyse. Comme nous n'utilisons pas d'analyse de flot de données, nous détournons tous les sauts conditionnels et même ceux qui ne nécessitent pas nécessairement une faute. Par exemple, au début de la fonction `VerifyPIN`, la valeur de `g_ptc` est vérifiée et doit être supérieure à zéro, comme le montre la figure 3.6 (ligne 2). Comme nous ne considérons qu'une seule tentative d'authentification, cette condition est toujours remplie pendant l'attaque. Par conséquent, les instructions de branchement liées à cette vérification ajoutent des contraintes inutiles en ajoutant une instruction typée *skip* et entraînent plus de fautes qu'il n'en faut vraiment.

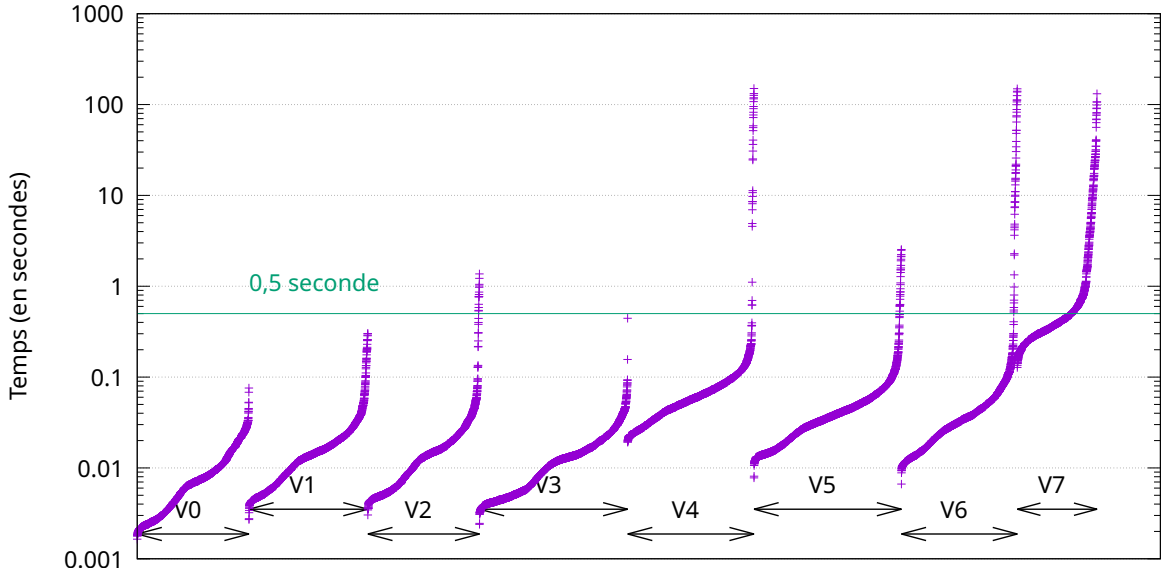


FIGURE 3.14 – Temps requis pour générer les chemins, pour chaque paramètre de faute considéré et par version de `VerifyPIN`, en utilisant la stratégie `R1 + R2` avec le rétrécissement des fautes activé

Temps d'exécution requis par l'analyse

Nous mesurons le temps d'exécution de notre implémentation afin d'évaluer ses performances. La figure 3.14 représente le temps nécessaire à SAMVA pour déterminer jusqu'à 30

chemins d'attaque distincts pour chaque version de `VerifyPIN` et pour chaque paramètre d'injection. Ces résultats n'incluent pas le temps de simulation servant à valider l'attaque. La stratégie utilisée pour ces mesures est la règle de typage `R1` avec le rétrécissement des fautes activé. Nous avons effectué nos mesures sur un processeur Intel i7-1165G7 à 2,3 GHz en allouant 4 cœurs physiques, sur lesquels des instances indépendantes de SAMVA sont lancées. Chaque instance de SAMVA est séquentielle, les temps rapportés sont donc indépendants du parallélisme de la machine. Nous obtenons des temps d'analyse relativement courts, la plupart étant inférieurs au seuil d'une demi-seconde. Pour les versions `V4` et `V7`, nous pouvons remarquer que les temps d'analyse peuvent augmenter de manière significative, jusqu'à 109 secondes et cela peut s'expliquer par l'utilisation de l'algorithme de positionnement des fautes qui utilise une méthode de retour en arrière. En effet, la majeure partie du temps d'analyse est en fait consacrée au positionnement des fautes et, selon le typage des instructions, il se peut que nous devions revenir souvent sur nos pas pour prouver la non-faisabilité d'un chemin d'attaque.

3.4 Travaux apparentés

Afin d'aider les évaluateurs de sécurité et les concepteurs de contre-mesures, différents outils de recherche de vulnérabilités et de chemins d'attaque ont été proposés. Cette section compare quelques outils déjà présentés dans la section 2.2 avec notre approche.

Potet et al. [14] proposent *Lazart*, un outil basé sur la modification du CFG au niveau LLVM-IR puis de l'exécution symbolique pour établir l'absence d'attaques basées uniquement sur des inversions de branches multiples. Cette approche permet de tester rapidement l'efficacité des contre-mesures logicielles, mais ne tient pas compte de l'agencement des instructions finales et nécessite donc une analyse complémentaire sur l'assembleur. Bien que les auteurs n'indiquent pas le temps requis par l'analyse, cette approche est intrinsèquement limitée par le moteur d'exécution symbolique qui fait face à l'explosion combinatoire du nombre de chemins ou des états dans le cas d'applications complexes avec des entrées symboliques qui ont un impact sur les accès à la mémoire ou le flot de contrôle.

Bréjon et al. [16] proposent l'infrastructure logicielle *RobustB* qui utilise la vérification formelle par la résolution SMT pour trouver des vulnérabilités dans un code binaire. Les fautes considérées sont soit un saut d'instruction unique, soit une corruption de registre unique. Les temps de vérification rapportés sur les mêmes benchmarks vont de quelques

minutes à quelques heures sans détails supplémentaires. En comparaison, notre approche nécessite moins de deux minutes dans le pire des cas et notre modèle d'attaquant englobe le modèle de faute du saut d'une seule instruction.

Given-Wilson et al. [95] proposent également une approche automatisée basée sur la vérification formelle pour trouver des vulnérabilités contre les attaques en fautes au niveau du code binaire. L'approche prend en compte les fautes permanentes et transitoires qui se traduisent par des codes mutants qui sont ensuite donnés à un vérificateur de modèle. Cette approche doit alors produire autant de codes mutants que le nombre de configurations de fautes à explorer. Elle ne peut pas s'adapter à des fautes multiples de différentes largeurs.

Werner et al. [13] étendent *CELTIC*, une solution basée sur la simulation permettant de rechercher des chemins d'attaque en considérant jusqu'à deux fautes. Les modèles de fautes utilisés sont inférés à partir d'expérimentations physiques, comme proposé précédemment par Dureuil et al. [12]. In fine, cette approche permet de sélectionner les paramètres d'injection de fautes pour réaliser des attaques. Toutefois, à l'instar d'autres approches basées sur la simulation [91], elle reste limitée par le nombre de fautes pouvant être injectées. Malgré le fait que la simulation soit plus efficace que les approches formelles pour analyser des programmes de grande taille, l'espace des configurations des fautes croît de manière exponentielle lorsque l'on considère des fautes multiples de largeurs variables. La convergence vers des configurations de fautes menant à des attaques réussies dépend de la stratégie d'exploration de l'espace des configurations de fautes. À notre connaissance, aucune approche de simulation actuelle ne peut gérer un grand nombre de fautes de ce type.

Ducousso et al. [15], [94] proposent *FASE*, une approche d'analyse de sécurité basée sur l'exécution symbolique au niveau ISA, conçue pour mieux passer à l'échelle face aux attaques multi-fautes. Contrairement aux autres méthodes symboliques nécessitant la création de mutants ou de bifurcations, *FASE* encode directement les effets des fautes dans des formules symboliques, qui sont ensuite résolues par des solveurs pour identifier les vulnérabilités potentielles. Plusieurs modèles de fautes sont pris en charge, notamment l'altération des données à différentes granularités (bit, octet et mot) et le saut d'instructions. Par rapport à SAMVA, cette solution inclut une analyse du flot de données en présence de fautes permettant de ne pas fauter des instructions parfois inutilement et donc d'identifier des chemins d'attaque nécessitant moins de fautes. *FASE* gère des attaques comprenant jusqu'à dix instructions fautées, tandis que SAMVA est capable de

considérer un plus grand nombre de fautes de largeurs variable.

En résumé, bien que SAMVA ne prenne actuellement en charge que les fautes de saut d'instruction, nous pensons qu'il est le premier outil basé exclusivement sur l'analyse statique capable de déterminer des chemins d'attaque nécessitant de nombreuses fautes de largeurs variables menant à des attaques réussies.

3.5 Conclusion

Dans ce chapitre, nous présentons SAMVA, une solution permettant d'évaluer les vulnérabilités d'un programme binaire contre les attaques par sauts d'instructions multiples. SAMVA est basé sur une analyse purement statique. Nous évaluons notre approche en déterminant les fautes nécessaires pour attaquer huit versions de programmes de vérification de codes PIN renforcés par diverses contre-mesures contre les fautes. Dans nos expériences, nous explorons de nombreuses capacités d'injection de fautes et les résultats montrent la capacité de SAMVA à trouver des chemins d'attaque réussis, même pour les implémentations les plus renforcées. Nous constatons également que notre approche passe bien à l'échelle en nombre de fautes, ce qui en fait un moyen efficace pour explorer un large éventail de configurations de fautes en un temps limité. Ces travaux ont été présentés au workshop international *COSADE* en 2023 [19].

Les résultats obtenus sont encourageants, mais le modèle de faute choisi n'est pas suffisamment précis pour réaliser des attaques réelles avec un banc d'injection. En effet, le saut d'instruction peut être le résultat de nombreux effets physiques comme le remplacement d'une instruction par une autre ou encore le rejeu d'une instruction. Davantage de contraintes relatives à la réalisation des fautes doivent être prises en compte durant l'analyse afin de trouver des chemins d'attaque exploitables en pratique. De plus, SAMVA permet de trouver la position et la largeur des fautes sur des instructions, cependant, un banc d'injection utilise des paramètres physiques comme l'instant et la durée des injections, ainsi que leurs intensités. Il est donc nécessaire de convertir les chemins d'attaque en configuration de fautes physiques et de synchroniser ces injections.

Ces différentes problématiques ont orienté nos travaux après cette première publication. Dans le chapitre suivant, nous présentons une méthode permettant de faire le lien entre notre analyse SAMVA et un banc d'injection afin de réaliser les attaques identifiées par l'analyse.

Chapitre 4

De la recherche de chemins d'attaque multi-fautes à l'injection automatisée de fautes

Les analyses de programme existantes cherchant à identifier des chemins d'attaque s'appuient sur des modèles de faute. Ces modèles permettent de représenter les effets des fautes à un niveau d'abstraction tel que l'ISA. Cependant, ils ne prennent pas en compte les contraintes physiques inhérentes à la réalisation des fautes induisant ces modèles. De ce fait, les chemins d'attaque identifiés ne sont pas toujours réalisables en pratique pour une plateforme d'injection et une cible donnée. Dans ce chapitre, nous souhaitons répondre à cette problématique en proposant SAMPLAI, une approche complète composée de ces trois éléments principaux : 1) une analyse statique extensible, basée sur SAMVA, capable de tenir compte, lors de la phase de recherche des chemins d'attaque, des capacités de l'attaquant ainsi que des conditions spécifiques nécessaires pour réaliser un saut d'instruction au niveau ISA ; 2) la conversion de ces chemins d'attaque en paramètres temporels pour l'injection de fautes ; et 3) la réalisation automatisée d'attaques en utilisant ces paramètres, combinés avec d'autres paramètres d'injection issus d'une calibration préalable du banc d'injection de fautes.

Dans le cadre de ces travaux, nous nous focalisons sur le modèle de rejeu d'instruction, une variante du saut d'instruction obtenue à l'aide de la plateforme d'injection de fautes TRAITOR. Ce banc d'injection permet d'altérer le signal d'horloge, induisant des fautes qui provoquent la réexécution de certaines instructions, permettant ainsi de sauter l'exécution des instructions situées après les instructions rejouées. Nous détaillons notre approche pour étudier les effets de ces injections de fautes et la manière dont nous en déduisons les conditions de réalisation du rejeu.

Nous évaluons notre chaîne d'injection de fautes en menant des attaques sur huit implémentations protégées d'un programme de vérification de code PIN. Les résultats obtenus

démontrent que notre analyse est capable d'identifier des chemins d'attaque, malgré les ajouts de contraintes relatives aux conditions des fautes. Des campagnes d'injections automatisées ont été menées sur trois cartes embarquées intégrant un microprocesseur Arm Cortex-M3. L'ensemble des programmes ciblés ont menés à des attaques réussies, nécessitant jusqu'à huit fautes induisant plus de 80 instructions sautées, avec une répétabilité élevée.

Ce chapitre est structuré comme suit : nous commençons par expérimenter des injections avec la plateforme TRAITOR sur notre cible pour étudier les effets des fautes au niveau ISA dans la section 4.1. Ensuite, dans la section 4.2, nous décrivons notre extension de l'analyse de programme permettant d'identifier des chemins d'attaque réalisables. Dans la section 4.3, nous présentons notre méthode d'évaluation et les résultats obtenus. Enfin, nous concluons dans la section 4.4.

4.1 Expérimentations du rejeu d'instruction

Cette section vise à comprendre les conditions de réalisation des fautes ayant pour effet le rejeu d'instruction. Cet effet de faute a déjà pu être observé par Alshaer et al. [56] et Claudepierre et al. [10], mais une caractérisation plus précise est nécessaire pour intégrer les critères de réalisation des fautes dans l'analyse de programme. Nous commençons par présenter notre dispositif expérimental comprenant la plateforme d'injection, l'appareil cible ainsi que la technique pour tester des codes. Ensuite, nous détaillons les observations que nous avons pu réaliser.

4.1.1 Dispositif expérimental

Nous utilisons le même dispositif expérimental durant nos observations du modèle de faute et lors de l'évaluation de notre méthode. Ce dispositif est composé d'un banc d'injection permettant de réaliser des fautes et d'un appareil cible que nous devons adapter à la plateforme d'injection.

Banc d'injection

Nous utilisons la plateforme d'injection de fautes TRAITOR (TRAnsportable glItch aTtack platfORm), implémentée sur un FPGA, afin de perturber le signal d'horloge d'un appareil cible [10]. TRAITOR génère le signal d'horloge de la cible et permet d'injecter des

fautes à n'importe quel cycle souhaité. Les fautes sont créées en combinant deux signaux d'horloge distincts générés par le FPGA. En déphasant les signaux de ces deux horloges, le signal d'horloge de sortie présente un front montant « cassé » à une amplitude choisie. Cette amplitude est paramétrable par l'utilisateur pour faire varier les effets des fautes.

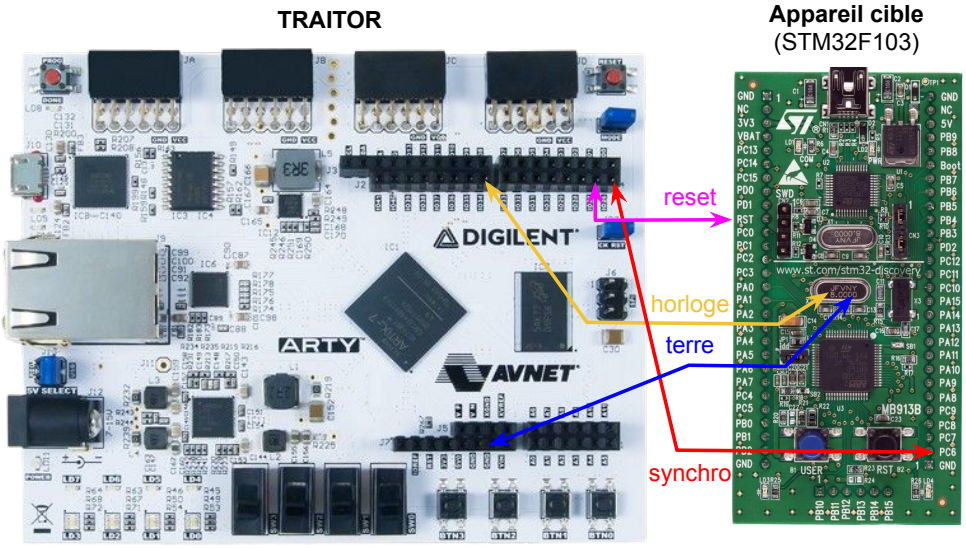


FIGURE 4.1 – Connexions entre TRAITOR et l'appareil cible

Afin d'évaluer ou d'attaquer un microcontrôleur, la source de son signal d'horloge, typiquement un quartz, est remplacée par la sortie de TRAITOR comme illustré dans la figure 4.1. Cette figure présente une vue d'ensemble du banc d'injection composé de TRAITOR et l'appareil cible. Pour faciliter l'injection de fautes avec TRAITOR, les mécanismes de contrôle de la phase (PLL) sur la cible sont désactivés.

Pour paramétrer les signaux d'horloge alternatifs utilisés pour attaquer la cible, l'utilisateur doit communiquer au FPGA les informations relatives aux fautes à injecter. Chaque injection de faute correspond à une séquence de cycles d'horloge perturbés. La configuration d'une faute requiert trois paramètres : le délai avant le début de la perturbation, exprimé en nombre de cycles ; la durée de la perturbation, également en nombre de cycles ; et l'amplitude du signal d'horloge. La figure 4.2 présente un signal d'horloge fauté en fonction des paramètres utilisés. Les valeurs maximales acceptées par TRAITOR pour ces paramètres sont encodés sur 16 bits. L'amplitude des signaux d'horloge fautés doit être programmée à l'avance et reste constante pour tous les cycles fautés de toutes les injections durant la même attaque. La valeur de l'amplitude correspond à une tension comprise entre 0 et 3,3 V (extrema d'un signal d'horloge commun).

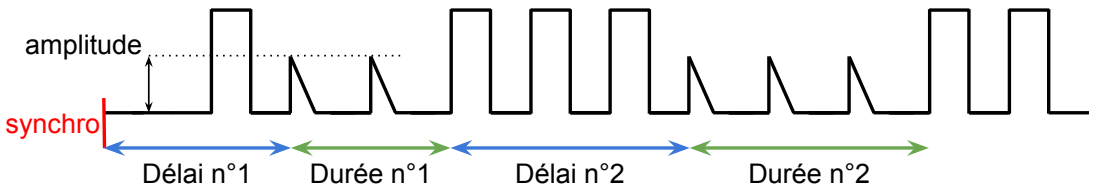


FIGURE 4.2 – Signal de sortie de TRAITOR

Appareil cible

Nous utilisons comme appareil cible un processeur Arm Cortex-M3 sur une carte STM32F100RB de la gamme STM32 Discovery. Ce processeur utilise un jeu d'instructions compressé, intégrant des instructions de tailles variables de 16 et 32 bits. Un processeur récupère les instructions en mémoire par blocs appelés mots, dont la taille est déterminée par sa microarchitecture. Un mot est considéré comme aligné si son adresse mémoire est un multiple de sa taille.

Pour le cas du Cortex-M3, les mots mémoire ont une taille de 32 bits, ce qui leur permettent de contenir soit deux instructions de 16 bits, soit une instruction de 32 bits, soit éventuellement une ou deux moitiés d'instructions de 32 bits. Son unité de *prefetch* charge jusqu'à trois mots d'instructions dans une mémoire tampon, en amont de leurs exécutions [107]. Lorsqu'un mot d'instruction du tampon est consommée, le processeur récupère les prochaines instructions avec une lecture de 32 bits. Il est nécessaire de connaître ce fonctionnement pour comprendre les effets des fautes.

La carte cible n'implémente aucune contre-mesure contre les attaques par injection de fautes. L'horloge peut être configurée en sélectionnant la source du signal, qu'il soit interne ou externe, avec la possibilité d'activer ou non la PLL et d'éventuels diviseurs. Pour attaquer cette carte avec TRAITOR, il est nécessaire de choisir une source de signal d'horloge externe (mode HSE pour *high-speed external oscillator*) et de remplacer le quartz initial par le signal d'horloge provenant de TRAITOR. Pour augmenter sa sensibilité aux fautes, la PLL est désactivée.

Une broche de la cible est connectée à TRAITOR afin de fournir un signal avant l'exécution du code à attaquer. Ce signal de déclenchement permet de synchroniser l'appareil cible avec la plateforme d'injection de fautes. Pour la configuration de TRAITOR, les délais avant les injections des fautes sont relatives à la réception de ce signal. Un léger décalage temporel entre la réception du signal et l'exécution du code visé peut exister et doit être déterminé au préalable au cours d'une étape de calibration. La broche RST est

aussi connectée à TRAITOR afin de pouvoir réinitialiser matériellement la cible si besoin.

Paramètres d'injection

Une attaque par injection de fautes nécessite de configurer le moyen d'injection pour chaque faute. Avec la plateforme TRAITOR, l'intensité d'une faute est déterminée par la valeur de l'amplitude du signal d'horloge. TRAITOR n'accepte qu'une seule valeur d'amplitude, qui sera donc appliquée uniformément à l'ensemble des fautes au cours de l'attaque. En plus de l'amplitude, chaque faute requiert des paramètres temporels : un délai, indiquant le nombre de cycles à attendre avant de commencer la perturbation et une durée, spécifiant le nombre de cycles que doit se maintenir la perturbation. Pour la première faute, le délai est relatif à la réception du signal de synchronisation. Pour les fautes suivantes, le délai est relatif à la fin de la faute précédente. Si un mécanisme de synchronisation n'est pas disponible, l'évaluateur doit identifier un signal compromettant qui peut être utilisé comme un signal déclencheur. Un signal compromettant peut être identifié à l'aide de méthodes d'analyse de canaux auxiliaires [108], notamment dans un contexte de boîte noire. En résumé, une attaque réalisée avec TRAITOR nécessite de définir une valeur d'amplitude, ainsi qu'un couple *(délai, durée)* pour chaque faute injectée.

Caractérisation préliminaire

Réaliser une étape préliminaire de caractérisation permet de déterminer les effets potentiels des fautes observables au niveau ISA, les paramètres d'injection permettant de réaliser des fautes utiles ainsi que les conditions de réalisation des fautes.

Pour cette caractérisation préliminaire, nous utilisons une suite de codes de test contenant des instructions assembleurs pour architecture Arm. Cette suite contient de nombreux tests, notamment sur les opérations arithmétiques, les lectures et écritures dans la mémoire et les branchements. Nous prenons également en compte les potentielles variations d'alignement et d'encodages des instructions. Hormis les instructions de saut, ces codes des instructions modifiant la valeur de leurs opérandes. Les registres sont choisis afin de rendre les rejeux d'instructions discriminants, de manière à pouvoir identifier quelles sont les instructions qui sont exécutées ou non, ou rejouées un certain nombre de fois.

SAMPLAI gère l'acquisition des données de caractérisation, en faisant varier les paramètres de fautes tels que l'amplitude du signal d'horloge fauté ainsi que l'instant et la durée de l'injection. Pour chaque ensemble de paramètres, la même injection est conduite

plusieurs fois afin de détecter d'éventuels changements de comportement et mesurer la reproductibilité des effets observés.

```
1  adds r0, r0, 11        # mot 1, cycle 50
2  subs r3, r3, 4
3  adds r1, r1, 13        # mot 2, cycle 52
4  subs r4, r4, 5
5  adds r2, r2, 17        # mot 3, cycle 54
6  subs r0, r0, 1
7  adds r3, r3, 19        # mot 4, cycle 56
8  subs r1, r1, 2
9  adds r4, r4, 23        # mot 5, cycle 58
10 subs r2, r2, 3
11 adds r0, r0, 29        # mot 6, cycle 60
12 subs r3, r3, 9
13 adds r1, r1, 31        # mot 7, cycle 62
14 subs r4, r4, 10
15 adds r2, r2, 37        # mot 8, cycle 64
16 subs r0, r0, 6
17 adds r3, r3, 41        # mot 9, cycle 66
18 subs r1, r1, 7
19 adds r4, r4, 43        # mot 10, cycle 68
20 subs r2, r2, 8
```

FIGURE 4.3 – Code de test en langage assembleur Arm comprenant 10 mots d'instruction composés d'une addition et d'une soustraction

À partir de l'état du système obtenu après l'exécution du code de test, SAMPLAI permet d'identifier automatiquement les effets des fautes. En nous basant sur l'hypothèse des auteurs de TRAITOR selon laquelle les fautes se manifestent exclusivement sous forme de sauts d'instruction ou de rejeux, nous avons développé un simulateur de fautes utilisant le moteur d'émulation de processeur Unicorn [109]. Ce simulateur découpe le code de test en mots mémoire et exécute de manière exhaustive toutes les combinaisons avec remplacement possibles de ces mots, mais en préservant leur ordre. Les répétitions de mots sont autorisées pour simuler le rejeu d'instructions. Étant donné que la simulation est exhaustive, nous limitons la taille des codes de test à dix mots mémoire.

À l'aide des simulations, nous construisons un tableau de correspondance entre les états possibles du processeur et les effets des fautes qui ont été identifiés. Après une attaque physique, l'état du système réel est sauvegardé et ce tableau de correspondance est consulté. Si l'état du système correspond a un état déjà rencontré lors des simulations,

alors on peut induire les effets des fautes qui ont mené à cet état. Plusieurs correspondances sont possibles pour un état donné selon le code de test utilisé. Nos tests sont discriminants et permettent d'identifier un état pour un effet de faute. Le reste de l'analyse est fait manuellement à l'aide des données issues des exécutions fautées.

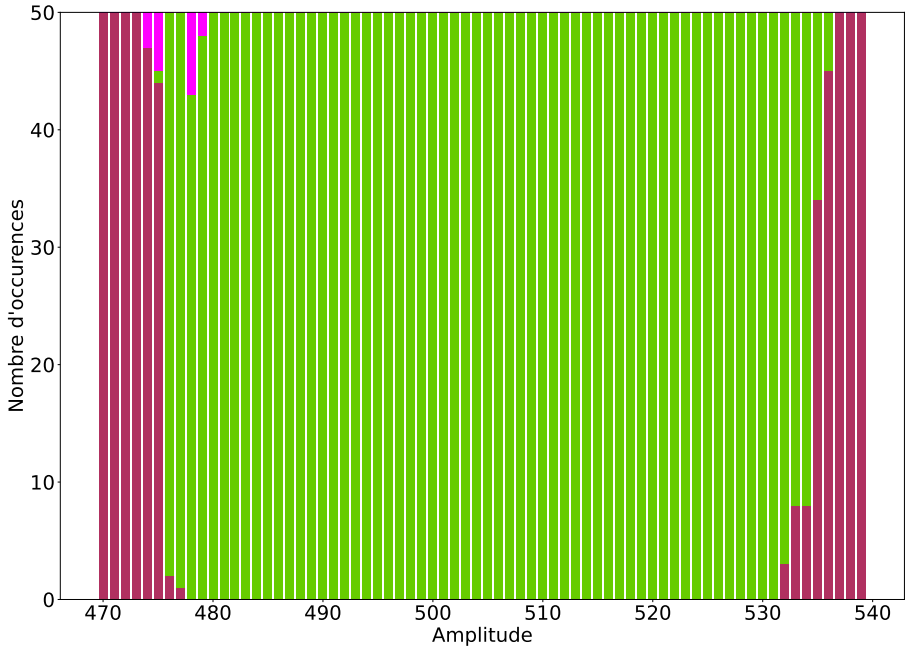
Dans la collection de codes de test, nous utilisons le code assembleur présenté dans la figure 4.3 afin d'illustrer notre méthode d'observation expérimentale des effets des fautes. Ce code est composé de dix mots mémoire contenant deux instructions 16 bits chacun. Un mot est composé d'une addition et d'une soustraction, avec une utilisation des registres permettant de discriminer les mots qui sont sautés ou rejoués. Comme pour tous les autres codes de test, cet extrait est entouré de deux séries d'instructions *NOP*, ne faisant rien et servant à isoler les effets des instructions environnantes. La fin du test se termine par un point d'arrêt avec une instruction *BKPT* (pour *breakpoint*), nous permettant de stopper l'exécution du programme puis de consulter l'état du système avec un débugueur après l'attaque. L'exécution du code de test commence après 50 cycles à partir du début de la fonction de caractérisation et chaque instruction prend 1 cycle pour s'exécuter.

4.1.2 Observations expérimentales

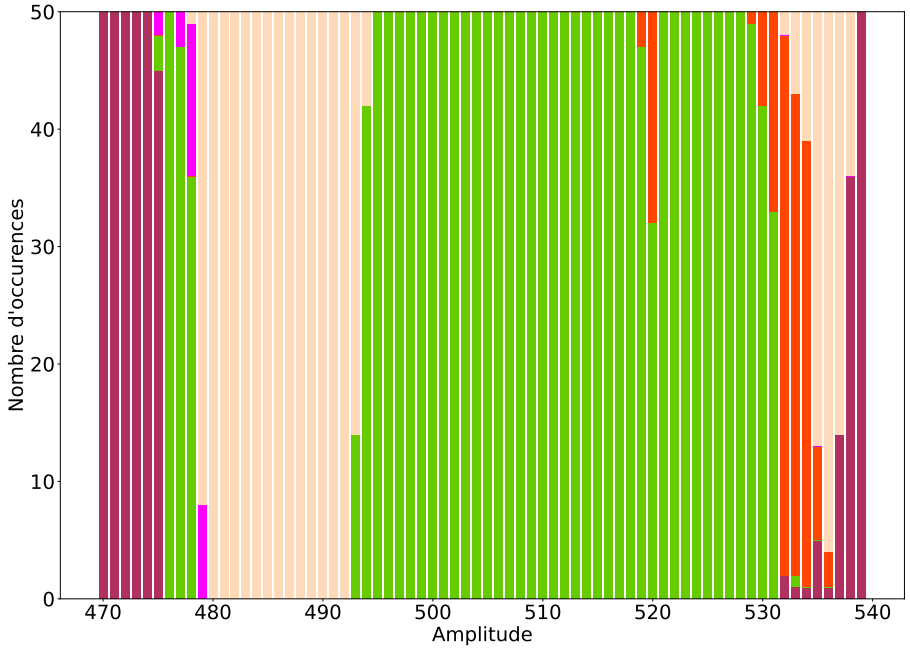
Pour l'acquisition des données servant à la caractérisation des fautes, nous réalisons 50 attaques avec une seule faute, pour chacun des paramètres de faute testés. L'ensemble des paramètres de faute testés est obtenu en faisant varier l'amplitude, le délai avant l'injection entre 48 et 68 cycles et enfin la durée de l'injection entre 1 et 16 cycles. Environ cinq attaques sont réalisées par seconde et l'acquisition des données présentées a duré 47 heures. Une attaque comprend la réinitialisation de l'appareil cible, la programmation de TRAITOR avec les paramètres de faute, une exécution fautée puis termine par une lecture de l'état du système et détermination automatique des effets de la faute.

Effets des fautes observés

À l'issue de ces expérimentations sur l'ensemble des codes, nous constatons que la plupart des fautes induisent un effet de saut ou de rejeu, avec une granularité au niveau du mot mémoire. Cela signifie que si un mot contient deux instructions de 16 bits, le saut ou le rejeu affectera ces deux instructions, et de même pour un mot contenant une seule instruction de 32 bits. Étant donné que le saut ou le rejeu d'instructions ne concernent que des mots mémoires, nous introduisons une notation pour représenter les effets des



(a) `durée = 2 cycles`



(b) `durée = 4 cycles`

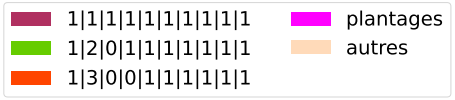


FIGURE 4.4 – Classification des effets des fautes pour `délai = 60 cycles`

fautes au niveau ISA. Cette notation se compose d'une suite de nombres séparés par des barres verticales. La position du nombre représente la position du mot mémoire du code de test et sa valeur définit le nombre d'exécutions du mot mémoire. Par exemple, pour un code de test contenant quatre mots mémoire, `1|1|0|1` signifie que le troisième mot a été sauté, tandis que `1|2|0|1` indique que le deuxième mot a été rejoué. Nous observons des effets des fautes qui sortent du modèle du rejeu d'instruction et cette notation permet aussi de les illustrer.

La figure 4.4 présente une classification des effets des fautes obtenus après une injection visant le code présenté dans la figure 4.3 avec une faute, selon l'amplitude du signal d'horloge fauté, pour deux cycles fautés (figure 4.4a) et quatre cycles fautés (figure 4.4b). Sur ces histogrammes, nous affichons avec la notation présentée précédemment les effets de rejeux, « plantages » si l'appareil ne répond plus après l'injection et « autres » si l'état obtenu ne correspond pas à un effet de rejeu d'instruction. Nous visons le deuxième mot du code de test et en faisant varier le délai avant l'injection, nous obtenons les effets souhaités avec un délai fixé à 60 cycles. Sachant que le deuxième mot du code de test commence après 52 cycles après le début de la fonction et que chaque instruction prend un cycle pour s'exécuter, nous induisons qu'il existe un délai entre la réception du signal de déclenchement (pin de synchronisation) et l'exécution du code qui vaut $60 - 52 = 8$ cycles.

Les résultats préliminaires indiquent que, pour les deux durées, les fautes injectées produisent un effet lorsque la valeur de l'amplitude est comprise entre 476 et 539. En dehors de cet intervalle, aucune perturbation n'est constatée et l'exécution semble se dérouler normalement. Sur la figure 4.4a nous constatons un rejeu du deuxième mot de manière très reproductible pour une valeur d'amplitude comprise entre 476 et 531. En revanche, la figure 4.4b affiche des effets catégorisés comme « autres », signifiant une plus grande diversité d'effets de fautes qui semble dépendre de la valeur de l'amplitude. D'après l'ensemble de tests que nous avons menés, les fautes avec une durée supérieure à 2 cycles semblent mener davantage à un effet de rejeu d'instructions lorsque l'amplitude est haute, proche de la borne supérieure, c'est-à-dire autour de 530. On constate que l'effet théorique du rejeu d'instruction est rencontré pour une amplitude comprise entre 532 et 537 (`1|3|0|0|1|...`). Les fautes peuvent aussi causer un plantage de l'appareil cible si la perturbation impact un composant critique du système ou si l'effet de la faute cause l'exécution d'une opération illégale comme une lecture ou une écriture à une adresse mémoire invalide.

Les nombreux effets rencontrés montrent que la perturbation du signal d'horloge peut causer divers effets matériels qui se traduisent au niveau ISA par des combinaisons de sauts et rejeux d'instructions, comme ceux catégorisés comme « autres » dans la figure 4.4b. Parfois plusieurs effets sont observables pour des paramètres de fautes identiques. Un phénomène de gigue (ou *jitter* en anglais) ou de fluctuation des signaux électroniques rend les fautes parfois difficiles à reproduire et s'explique par des imprécisions expérimentales de plusieurs sources comme la sortie de l'horloge de TRAITOR, le mécanisme de synchronisation entre la cible et TRAITOR, l'exécution du programme de l'appareil cible ou enfin des perturbations liées à des injections de fautes précédentes.

Pour identifier des chemins d'attaque compatibles avec TRAITOR, nous adaptons notre analyse pour des fautes provoquant un effet de rejeu tel que prédit par le modèle. Par conséquent, il est nécessaire d'identifier les paramètres d'injection de fautes ayant la plus haute probabilité de produire l'effet souhaité. Dans la suite de cette section, nous analysons l'impact de l'amplitude et de la durée de l'injection sur les effets des fautes.

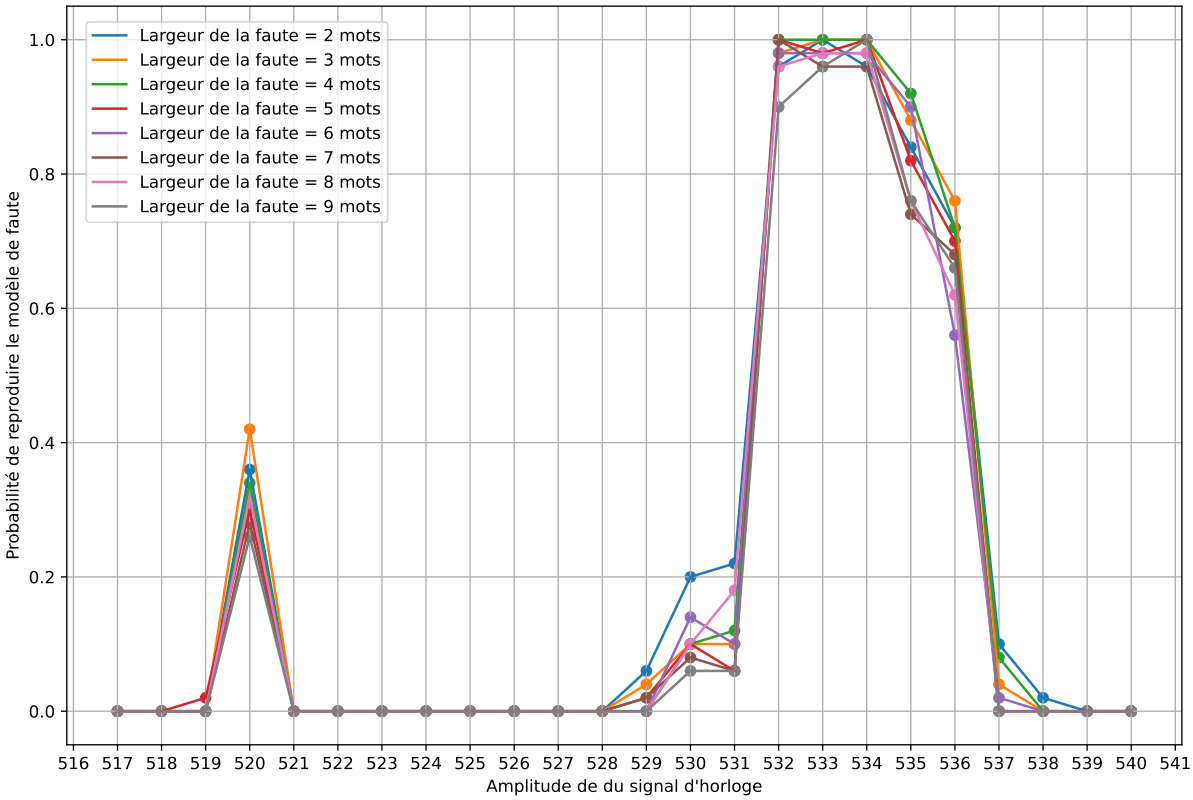


FIGURE 4.5 – Probabilité de l'effet de rejeu d'une faute en fonction de l'amplitude

Amplitude des fautes

Le paramètre de l'amplitude du signal d'horloge fauté influence sensiblement les effets des fautes au niveau de l'ISA. Au travers de plusieurs tests préliminaires, nous observons que le modèle de rejeu d'instruction est plus fréquent lorsque l'amplitude est configurée près de la borne supérieure, c'est-à-dire avec une valeur comprise entre 520 et 540. Au-delà de cette plage, aucun effet observable n'est détecté, tandis qu'en deçà, les effets des fautes deviennent plus imprévisibles.

La figure 4.5 montre les différents taux de reproductibilité des effets de fautes en fonction de l'amplitude utilisée avec un délai fixé à 60 cycles. Nous considérons des fautes de diverses largeurs et suivant le modèle de rejeu d'instruction. La probabilité de reproduction du modèle de faute est déterminée à partir des résultats de 50 attaques par paramètre évalué sur le code de test présenté dans la figure 4.3. Nous constatons que la reproductibilité des fautes est similaire pour toutes les largeurs, à l'exception des fautes ayant une largeur de deux mots, qui présentent une reproductibilité plus élevée. Pour une amplitude comprise entre 532 et 536, la probabilité d'observer un effet de rejeu, toutes largeurs confondues, est supérieure à 0,9. Ces résultats montrent une reproductibilité élevée des fautes, mais à condition que les paramètres de fautes soient bien choisis. Ces effets de fautes sont également reproductibles en injectant la faute un cycle plus tard que le délai précédemment présenté, c'est-à-dire pour un délai fixé à 61 cycles. Toutefois, les résultats montrent une probabilité moins élevée : autour de 0,8 pour une amplitude comprise entre 532 et 534 et environ 0,3 pour une amplitude comprise égale à 535 ou 536. Ces différences démontrent la sensibilité des paramètres de fautes sur les effets observables des fautes.

Durée des injections

La largeur d'une faute, c'est-à-dire le nombre de mots mémoire sautés en raison de l'effet de rejeu d'instructions, est directement liée à la durée de la perturbation du signal d'horloge. La figure 4.6 illustre le lien entre la durée de l'injection et le nombre de rejeux du mot d'instruction ciblé. Les durées d'injection testées varient de 1 à 16 cycles. Chaque point de la figure est annoté avec l'amplitude utilisée pour obtenir l'effet et sa couleur indique la probabilité d'obtenir cet effet (cf. l'échelle à la droite de la figure). Le mot ciblé contient deux instructions 16 bits et prend 2 cycles pour s'exécuter.

Nous observons une corrélation entre la largeur d'une faute et la durée de l'injection : plus le nombre de rejeux d'un mot mémoire est élevé, plus la durée de l'injection doit être

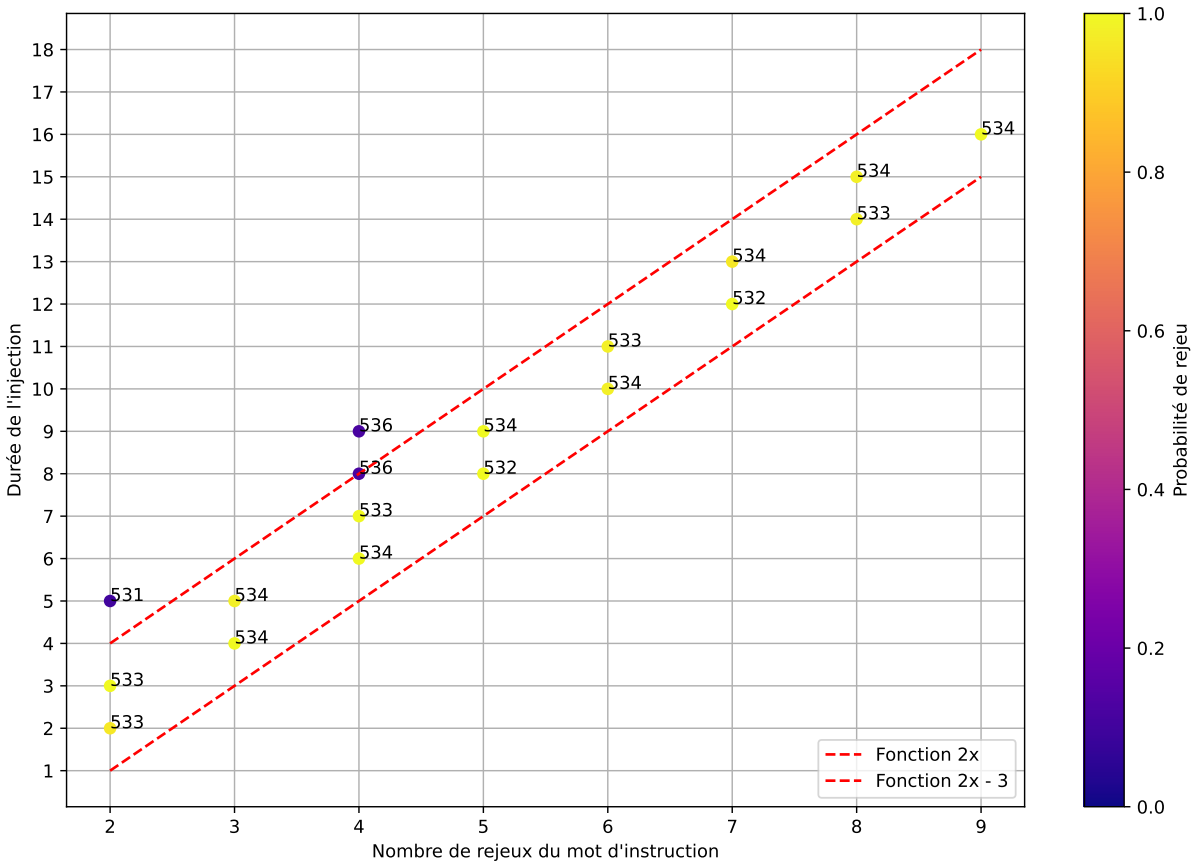


FIGURE 4.6 – Nombre d'exécutions du mot ciblé en fonction de la durée de l'injection

longue. Cependant, une injection prolongée peut entraîner de nombreux effets. Si l'on considère x comme le nombre de rejeux, nous estimons qu'une durée d'injection comprise entre $2x-3$ et $2x$, comme affiché par les droites rouges dans la figure, permet de configurer TRAITOR afin d'obtenir les effets souhaités en limitant l'espace des paramètres.

À partir de ces observations, nous étendons SAMVA pour prendre en compte les conditions de réalisation de l'effet de rejeu de mot aligné, puis nous proposons une méthode pour déterminer automatiquement des paramètres d'injection afin de réaliser physiquement les chemins d'attaque identifiés.

4.2 Méthode SAMPLAI : de l'analyse à l'injection

Notre approche est une extension de l'analyse statique SAMVA que nous avons présentée dans le chapitre 3. SAMVA permet d'identifier des chemins d'attaque multi-fautes,

avec des fautes de largeur variable ayant pour effet le saut d'instructions. L'extension que nous proposons permet d'exprimer les contraintes relatives aux conditions de réalisation de fautes causant des rejeux d'instructions ainsi que d'automatiser la conversion des chemins d'attaque théoriques en paramètres d'injection de fautes physiques. Cette section débute par présenter une vue d'ensemble de notre approche, puis nous présentons l'extension de l'analyse et la technique de détermination des paramètres de fautes.

4.2.1 Vue d'ensemble

L'approche SAMPLAI vise à faciliter les attaques par injection de fautes en identifiant des chemins d'attaque plus réalistes en fonction du moyen d'injection utilisé, puis en traduisant ces chemins théoriques en paramètres d'injection concrets. La figure 4.7 présente un aperçu du déroulement de ces étapes. Tout d'abord, une analyse du binaire de la cible est effectuée à l'aide d'une version étendue de SAMVA, capable de prendre en compte les conditions de réalisation des fautes (①). Ensuite, ces chemins d'attaque, définissant les instructions à fauter ou à exécuter, sont convertis en un ensemble de paramètres d'injection physiques (②). Cette conversion repose une mesure des temps d'exécution (en cycles) des instructions à exécuter pour un chemin d'attaque donné. Cette mesure est réalisée directement sur l'appareil cible en utilisant des mécanismes d'instrumentation et des registres de débogage. Ces mesures combinées à des heuristiques de sélection de paramètres, permettent de déterminer un ensemble de paramètres d'injection susceptibles de mener à une attaque réussie (③). À l'issue d'une attaque réalisée avec les paramètres définis à l'étape précédente, nous utilisons au choix, une communication série ou une lecture de la mémoire à l'aide d'un outil d'instrumentation pour connaître l'état du système et savoir si l'attaque a réussi ou échoué (④). Nous décrivons plus en détail chacune de ces étapes dans la suite de cette section.

4.2.2 Détermination des chemins d'attaque

SAMVA considère le saut d'instruction au niveau ISA sans tenir compte des conditions microarchitecturales permettant la réalisation du modèle de faute de saut d'instruction. Pour obtenir des chemins d'attaque qui peuvent être exploités dans la pratique, nous ajoutons la possibilité de les exprimer sous la forme de nouveaux modules SAMVA, s'intégrant ainsi dans l'analyse existante.

Nous considérons le modèle du saut d'instruction résultant d'un rejeu des instructions

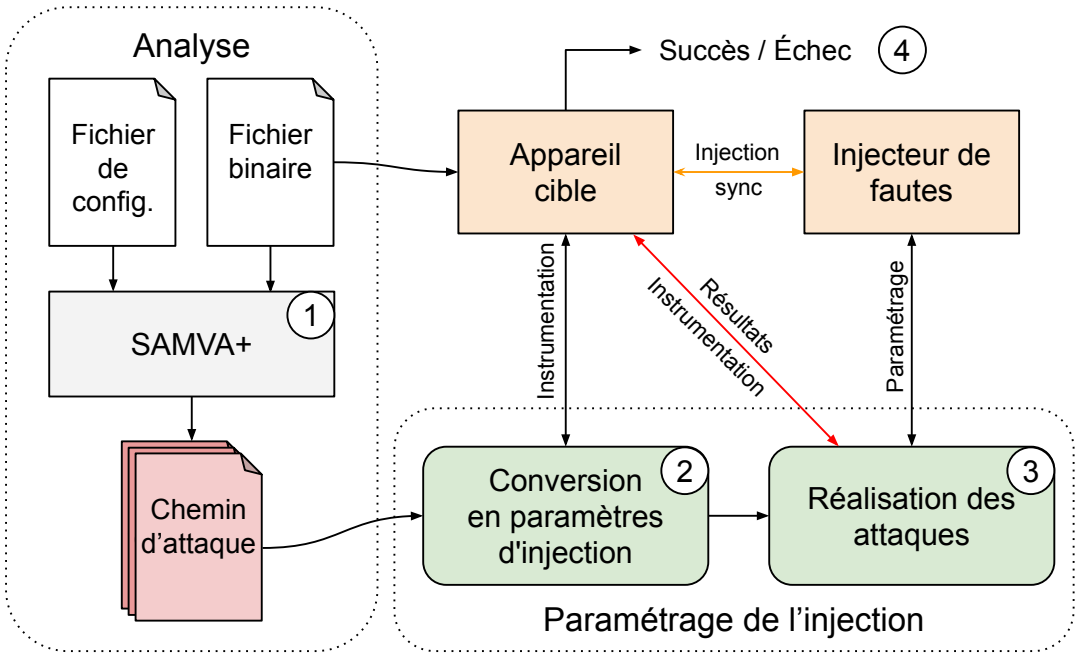


FIGURE 4.7 – Vue d'ensemble de l'approche SAMPLAI

d'un mot mémoire, comme observé par diverses techniques d'injection de fautes [5], [10], [56] puis confirmé par nos expérimentations préliminaires de la section 4.1. Selon ce modèle, les fautes se manifestent au niveau ISA sous la forme d'une réexécution d'un mot d'instruction aligné, tandis que le pointeur d'instruction continue à s'incrémenter. Du fait que les instructions contenues dans le mot sont rejouées à la place des instructions qui sont positionnées après, la faute provoque un saut de plusieurs instructions. Une explication probable de cet effet est que la faute perturbe l'étage de *fetch* du pipeline du processeur, l'empêchant de récupérer les nouvelles instructions à exécuter. Le processeur se retrouve alors à réexécuter les instructions déjà présentes dans le tampon mémoire, tout en incrémentant son pointeur d'instruction interne lors des rejeux.

Toutefois, les rejeux peuvent causer un plantage si le mot contient une instruction qui n'est pas complète ou si l'état du système est incohérent après la faute. De ce fait, nous présentons, ci-après, deux nouvelles contraintes relatives au positionnement des fautes qui sont ajoutées à SAMVA : la première impose une contrainte d'alignement supplémentaire pour les instructions rejouées et la largeur des fautes. La seconde restreint les instructions rejouées pour qu'elles n'aient pas d'effet sur le contexte du programme lorsqu'elles sont répétées. Ensuite, nous expliquons la façon dont les chemins d'attaque trouvés avec ces

extensions sont validés avant de démarrer un processus de paramétrage et d'injection de fautes.

Alignement des instructions

Les jeux d'instructions intégrant des instructions de longueur variable sont couramment utilisés dans les systèmes embarqués pour réduire la taille des binaires. Par exemple, l'ISA ARMv7-M Thumb, utilisé par le processeur présenté dans la section 4.1, ainsi que les ISA RISC avec l'extension C qui ajoute le support des instructions 16 bits, illustrent cette approche. Pour ces ISA, les binaires sont composés d'un mélange d'instructions de 16 et 32 bits. Comme mentionné précédemment, un processeur récupère les instructions par mots mémoire à des adresses multiples de 32 bits. Cependant, les instructions ne sont pas nécessairement alignées sur les mots mémoire, ce qui implique qu'un mot peut contenir une moitié d'une instruction 32 bits. Dans ce cas de figure, le rejeu du mot provoquerait un plantage, car cette moitié ne serait pas considérée comme une instruction valide. Il existe des cas où des instructions désalignées peuvent former une instruction 32 bit valide [56], mais ils ne sont pas considérés dans cette analyse. Ainsi, notre analyse contraint l'analyse pour qu'une faute démarre uniquement après l'exécution d'un mot mémoire aligné contenant soit une instruction 32 bit, soit deux instructions 16 bit. Ainsi, nous nous assurons que les mots pouvant être rejoués comprennent que des instructions complètes.

Idempotence des instructions

Une instruction est définie comme *idempotente* si elle peut être exécutée plusieurs fois tout en produisant systématiquement le même résultat. Cela implique qu'une instruction idempotente ne modifie pas ses opérandes. L'analyse considère qu'un mot peut être rejoué si les instructions qu'il contient sont toutes idempotentes. Cette contrainte permet d'éviter la répétition d'instructions pouvant produire des plantages, en particulier les instructions de chargement et de stockage qui mettent à jour l'une de leurs entrées (par exemple, les instructions `PUSH` ou `POP`). Ces instructions doivent être également alignées en mémoire comme indiqué ci-avant.

Validation des chemins d'attaque

Comme évoqué dans le chapitre 3, SAMVA n'utilise pas d'analyse de flot de données et en conséquence, des chemins d'attaque identifiés par l'outil peuvent entraîner un plantage en raison d'accès mémoire invalide lors de leur exécution par exemple. Pour pallier cette limitation, l'analyse simule les chemins d'attaque pour filtrer les chemins qui n'aboutissent pas à une attaque réussie ou à un plantage du système. Initialement, SAMVA utilise une version modifiée du simulateur *gem5* [102] pour simuler les sauts d'instructions. Dans cette extension, nous proposons deux nouvelles méthodes de validation des chemins d'attaque, capables de simuler des rejeux d'instructions.

La première méthode repose sur le moteur d'émulation au niveau ISA *Unicorn* [109]. En utilisant les informations issues de l'analyse du programme, le code qui est exécuté durant l'attaque est recopié dans la mémoire virtuelle de l'émulateur. Pour reproduire les effets des fautes, à chaque injection, le contenu des mots d'instructions devant être sautés est substitué par celui du mot à rejouer. Cela revient à remplacer les instructions devant être sautées par celles qui doivent être réexécutées. Cette méthode permet de filtrer rapidement les chemins d'attaque identifiés par l'analyse qui mènent à un plantage. Cependant, l'émulation n'intègre pas d'unité de protection mémoire (MPU) comme sur l'appareil cible. Par conséquent, des écritures à des adresses mémoire interdites sur la cible peuvent ne pas mener à un plantage lors de l'émulation. Pour une validation des chemins d'attaque plus proche de l'exécution réelle, qui ne considère pas seulement l'ISA, nous proposons une seconde méthode basée sur de l'instrumentation automatisée. Cette approche consiste à exécuter le code analysé directement sur l'appareil cible en simulant les effets des fautes à l'aide d'un débogueur. Pour simuler le rejeu d'instructions, le programme est exécuté en mode pas à pas. Lorsque des instructions doivent être sautées, le pointeur d'instruction (`PC`) est manipulé de manière à réexécuter le mot d'instruction précédent autant de fois que de mots doit être sauté. Bien que cette méthode soit plus lente, reproduire les fautes directement sur l'appareil cible permet de détecter plus précisément les potentiels plantages en amont des attaques physiques.

4.2.3 Conversion des chemins en paramètres d'injection

Mener une attaque par injection de fautes à partir d'un chemin d'attaque est une tâche difficile, principalement en raison de la complexité à convertir des sélections d'instructions à fauter en paramètres d'injection physiques concrets. La méthode que nous proposons

pour effectuer cette conversion présuppose qu'il existe un moyen de synchroniser précisément le début de l'injection avec l'appareil cible, comme expliqué dans la section 4.1.1. Dans le cas contraire, l'évaluateur devra recourir à des techniques d'analyse de canaux auxiliaires pour identifier un signal compromettant permettant d'assurer cette synchronisation avec la plateforme d'injection [108]. Dans la suite, nous expliquons comment nous convertissons les chemins d'attaque en paramètres temporels d'injection, avec le délai et la durée pour chacune des fautes.

Mesure du temps d'exécutions des instructions

Notre méthode de conversion d'un chemin d'attaque en paramètres d'injection temporels repose sur la mesure précise du temps d'exécution de chaque instruction devant être exécutée à partir de la réception du signal de synchronisation. Ces mesures permettent ensuite de déterminer le délai avant l'injection de chaque faute et de définir la durée de chaque injection, de manière à couvrir toutes les instructions devant être sautées. Le temps d'exécution des instructions peut être mesuré de différentes manières, en fonction des capacités de l'appareil ciblé, par exemple via un débogueur, une sonde JTAG, ou encore un minuteur système (*system timer*). Comme le démontrent les observations expérimentales, une précision au cycle près permet d'identifier les paramètres de fautes avec de meilleurs taux de reproductibilité.

Nous basons notre procédure de mesure sur une combinaison d'instrumentation du code sur l'appareil cible et de l'utilisation d'un registre de débogage. Les architectures Arm fournissent un registre de débogage optionnel nommé DWT (pour *Data WatchPoint and Trace Unit*). Ce registre spécial permet de placer des points de contrôle matériel, de sonder certains états internes du microprocesseur, et notamment, de fournir un compteur de cycles d'horloge (*CYCCNT*). En exécutant le chemin d'attaque pas à pas et en lisant le compteur d'horloge après chaque instruction, il devient possible de mesurer le temps d'exécution des instructions dans des conditions identiques à celles de l'attaque réelle. Nous utilisons OpenOCD et GDB pour interagir avec le programme en cours d'exécution.

Un chemin d'attaque fournit une trace théorique de l'attaque sous la forme d'une liste d'adresses, en spécifiant pour chacune si l'instruction qui s'y rapporte doit être exécutée ou sautée. À partir de cette liste, il est possible de déduire les multiples séquences d'instructions à exécuter. Nous définissons adr_n^d et adr_n^f comme respectivement l'adresse de la première et la dernière instruction pour la n-ième séquence d'instructions à exécuter. Les mesures sont effectuées en itérant la liste des séquences selon les étapes suivantes :

i) Avant de mesurer la première séquence, un point d'arrêt est configuré à la première adresse de la première séquence, soit adr_1^d, puis l'appareil cible est réinitialisé. Une fois le point d'arrêt atteint, celui-ci est supprimé et la mesure peut commencer.

ii) Une lecture initiale du compteur de cycles est réalisée, l'instruction à l'adresse courante (`PC`) est exécutée en utilisant la commande de pas à pas du débogueur (*step*), puis une nouvelle lecture du compteur de cycles est réalisée. La différence entre ces deux mesures correspond au temps d'exécution de l'instruction. Le `PC` est automatiquement incrémenté par le processeur selon la taille de l'instruction exécutée.

iii) Si la valeur du précédent `PC` correspond à adr_n^f, cela signifie que toutes les durées d'exécution des instructions de la séquence ont été mesurées et que les prochaines instructions sont censées être sautées. Par conséquent, le `PC` est directement mis à jour avec la valeur de adr_{n+1}^d pour simuler le saut d'instructions. Autrement, si l'instruction exécutée n'est pas la dernière de sa séquence, alors une nouvelle mesure est réalisée comme décrit pas le point précédent.

iv) Le processus de mesure se poursuit jusqu'à ce que le `PC` atteigne l'adresse de fin de la dernière séquence.

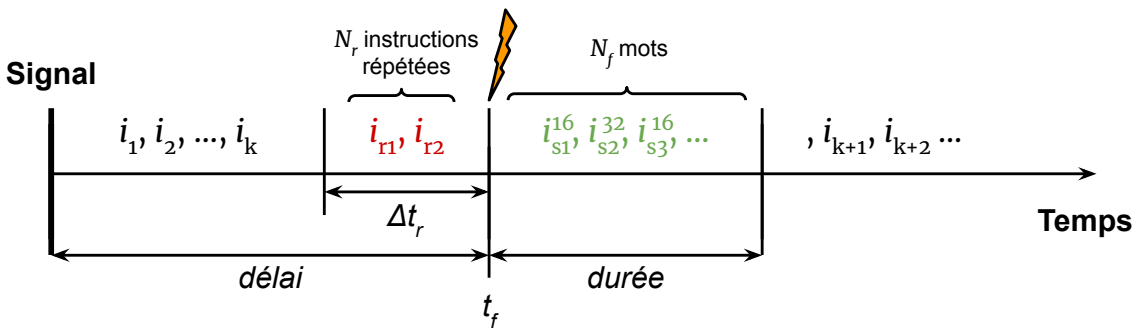


FIGURE 4.8 – Illustration du délai et de la durée d'une faute en fonction de la durée d'exécution Δt_r du mot qui précède la faute et de l'instant de l'injection t_f

Paramétrage des injections de fautes

Chaque faute requise pour l'attaque nécessite des paramètres temporels exprimés en cycles d'horloge : le délai avant l'injection et la durée de l'injection. Sur la plateforme d'injection TRAITOR, le délai de la première faute est mesuré par rapport à la réception du signal de synchronisation, tandis que pour les fautes suivantes, le délai est mesuré par

rapport à la fin de la dernière injection. Ainsi, le délai d'une faute correspond au nombre de cycles nécessaires pour exécuter la séquence d'instructions précédant l'injection. Nous déterminons un délai initial pour chaque injection n, noté $délai_n$ en utilisant les mesures des temps d'exécution des séquences d'instructions réalisées préalablement. Cependant, des variations temporelles peuvent survenir en raison du phénomène de gigue ou des effets des fautes précédentes. Ces variations, difficiles à anticiper, sont prises en compte en faisant varier dans un petit intervalle le délai initial à l'aide d'une nouvelle variable. Pour chaque injection n, nous définissons un intervalle de délais sous la forme $délai_n \pm k$, où k représente la marge d'approximation maximale.

La durée d'une injection correspond au temps d'exécution nécessaire pour rejouer le mot d'instructions qui précède la faute suffisamment de fois pour couvrir les instructions à sauter. Soit Δt_r, le nombre de cycles requis pour exécuter les N_r instructions contenues dans le mot mémoire avant l'injection, et N_f le nombre de mots mémoire à sauter, comme illustré dans la figure 4.8. Selon le modèle de rejeu d'instructions, la durée de l'injection, permettant de sauter les instructions désirées devrait être égale à $\Delta t_r \times N_f$. Cependant, rejouer des instructions peut entraîner des variations de temps d'exécution dues à des facteurs tels que l'occupation du bus de données ou les blocages de pipeline (*stall*). De plus, des phénomènes de gigue peuvent également affecter la durée. Nous avons observé ces variations temporelles lors de nos expérimentations préliminaires comme décrit dans la section 4.1.2. Pour gérer ces incertitudes, nous encadrons la durée d'injection n entre une borne inférieure et une borne supérieure : $1 \leq durée_n^{min} \leq durée_n \leq durée_n^{max}$. Par exemple, la calibration de notre plateforme d'injection a permis de définir des bornes d'encadrement telles que $durée_n^{min} = \Delta t_r \times N_f - 3$ et $durée_n^{max} = \Delta t_r \times N_f$, comme illustré dans la figure 4.6.

In fine, nous construisons deux ensembles de valeurs pour chaque faute n : l'un pour les paramètres de délai noté $D_n^{é}$ et l'autre pour la durée noté D_n^{u}. En réalisant le produit cartésien de ces deux ensembles, nous obtenons un ensemble noté P_n tel que $P_n = D_n^{é} \times D_n^{u}$. Cet ensemble P_n contient tous les couples *(délai, durée)* pour une faute donnée, potentiellement capables de provoquer la faute souhaitée.

Automatisation des attaques

En raison de la nature stochastique des effets des perturbations physiques, réaliser une attaque par injection de fautes requiert de tester plusieurs configurations de paramètres, et pour chacune, d'effectuer potentiellement plusieurs répétitions. Cependant, lorsqu'un

ensemble de paramètres est défini pour chaque faute, le nombre de configurations possibles pour une attaque augmente de manière exponentielle avec le nombre de fautes requises. Pour réduire la quantité d'injections à réaliser, nous introduisons deux limites à définir avant l'attaque concernant le nombre de configurations à tester et le nombre de répétitions à effectuer. Ces deux limites sont configurables par l'évaluateur en fonction du temps alloué pour réaliser l'attaque et de la précision du banc d'injection utilisé.

Nous construisons la liste L, une collection ordonnée des configurations de paramètres de fautes pour une attaque contenant i fautes. Cette liste est définie comme le produit cartésien des ensembles de paramètre de chacune des fautes :

$$L = \prod_{n=0}^{i} P_n = P_0 \times P_1 \times \cdots \times P_i$$

Si l'attaque comporte de nombreuses fautes, la taille de cette liste peut devenir très conséquent, voire impossible à tester dans son intégralité. Pour remédier à cela, nous construisons la liste finale en réalisant le produit cartésien des paramètres des fautes de façon incrémentale.

$$\begin{aligned} L &= P_0 \\ L &= L \times P_1 \\ &\vdots \\ L &= L \times P_i \end{aligned}$$

Ainsi, la taille de la liste des configurations L est vérifiée à chaque itération du produit cartésien. Si la taille est supérieure à la limite fixée par l'évaluateur, la liste est mélangée aléatoirement puis tronquée à la taille de cette limite, puis le calcul se poursuit à la prochaine itération. Le fait de mélanger permet d'obtenir un sous-ensemble représentatif pour la taille permise.

Un paramètre de faute valide peut ne pas produire l'effet escompté à chaque tentative. Ainsi, plusieurs répétitions de l'attaque sont possibles lorsqu'une configuration est testée. Le nombre de répétitions est défini par l'évaluateur avant de lancer la campagne d'injection. De plus, diverses amplitudes de faute peuvent être envisagées pour chaque attaque. Ces amplitudes, déterminées lors d'une étape de calibration préalable, doivent aussi être spécifiées en amont. Chaque amplitude est donc testée pour chaque configuration d'attaque.

Pour chaque chemin d'attaque, le processus d'automatisation des injections de fautes

se déroule en plusieurs étapes :

i) L'appareil cible est reprogrammé avec le binaire à attaquer.

ii) L'ensemble des configurations d'attaque est généré, en tenant compte de l'éventuelle limitation de taille imposée par l'évaluateur.

iii) Les attaques sont effectuées via trois boucles imbriquées : pour r allant de 0 jusqu'au nombre maximal de tentatives spécifié par l'évaluateur, pour chaque amplitude dans l'ensemble des amplitudes, et enfin pour chaque configuration dans l'ensemble des configurations de fautes, la plateforme d'injection est programmée avec l'amplitude et la configuration correspondantes.

iv) L'appareil cible est réinitialisé et l'exécution du programme est lancée.

v) Lorsque l'exécution du programme est terminée, l'état du système et le résultat sont récupérés puis journalisés.

vi) Le résultat est ensuite analysé pour déterminer si l'attaque a réussi ou non.

vii) En cas de succès, la campagne d'injection pour ce chemin d'attaque est arrêtée.

4.3 Évaluation

Nous évaluons notre méthode SAMPLAI en testant la sécurité des programmes de vérification de code PIN issus de la suite de benchmarks FISCC [12]. Nous commençons par présenter notre dispositif expérimental, puis nous présentons les résultats obtenus. Enfin, nous proposons quelques suggestions pour concevoir des contre-mesures plus efficaces, basées sur les résultats de nos expérimentations.

4.3.1 Dispositif expérimental

Pour réaliser les expérimentations, nous utilisons le même banc d'injection que celui utilisé durant notre processus de caractérisation, comme détaillé dans la section 4.1. Les fautes sont introduites par altération du signal d'horloge avec la plateforme TRAITOR et la cible est une carte STM32F1 intégrant un microprocesseur Arm Cortex-M3.

Programmes attaqués

Pour les expériences, nous ciblons les programmes de vérification du code PIN du projet FISCC [12]. Cette collection de logiciels contient huit implémentations de `VerifyPIN`, une

implémentation naïve et sept autres incorporant différents ensembles de contre-mesures logicielles. Les programmes comparent le code PIN fourni par l'utilisateur et le code PIN de la carte et accordent l'authentification s'ils sont identiques. Ces programmes de test ont été utilisés lors de l'évaluation de SAMVA et sont décrits plus en détail dans la section 3.3 du chapitre 3.

Les huit versions de `VerifyPIN` sont compilées pour l'ISA ARMv7-M implémentée dans les processeurs Cortex-M3. Toutes les optimisations du compilateur sont désactivées (`-O0`) pour éviter la détérioration des contre-mesures logicielles intégrées. Puisque la recherche du chemin d'attaque est sensible à l'alignement des instructions, nous générons une variante supplémentaire pour chaque version de `VerifyPIN`. Dans cette variante, une instruction `NOP` (*no-operation*), ne faisant rien, est insérée après le prologue de la fonction ciblée afin de décaler toutes les adresses du binaire de 2 octets. En définitive, nous compilons deux binaires pour chacune des 8 versions de `VerifyPIN`, ce qui donne un total de 16 binaires.

Configuration de l'analyse

Nous exploitons l'extension de SAMVA pour déterminer les chemins d'attaque pour chacun des 16 binaires décrits précédemment. L'analyse prend en compte deux paramètres : 1) la spécification de l'exploit et 2) les capacités de l'attaquant. Pour la spécification de l'exploit, nous identifions manuellement les instructions qui doivent être atteintes ou évitées conformément à l'objectif de l'attaque. Les attaques contre les programmes `VerifyPIN` sont définies pour atteindre les instructions qui fixent la variable d'authentification à « vrai » tout en évitant l'exécution de la fonction de contre-mesure. Les capacités de l'attaquant sont définies par rapport aux observations pratiques réalisées durant la calibration de la plateforme d'injection. La distance minimale entre deux injections de fautes est alors fixée à 1 mot. La largeur minimale d'une faute est de 1 mot et la largeur maximale est de 10 mots.

Pour réduire le nombre de chemins d'attaque qui conduisent à un plantage durant l'attaque, nous appliquons, comme indiqué dans le chapitre 3, deux règles supplémentaires de typage des instructions (`R1 + R2`). La première indique que les instructions qui mettent à jour le registre du pointeur de pile (registre `SP`) ne peuvent pas être fautées. Cette règle protège le pointeur de pile et garantit sa cohérence. La deuxième règle empêche le saut des instructions de branchement utilisant le registre de lien (registre `LR`), comme l'instruction `bx lr` qui est généralement utilisée pour le retour de la fonction.

Grâce à cette règle, le flot de contrôle original relatif aux appels de fonction est préservé. Les contraintes relatives aux positions permises de début et de fin des fautes, ajoutées pour le support du modèle de rejeu d'instructions, réduisent considérablement l'espace de recherche des positions possibles pour un chemin candidat. Contrairement à la version initiale de SAMVA, notre approche consiste à explorer exhaustivement toutes les positions potentielles des fautes pour un chemin. Ainsi, pour un chemin candidat, qui est une de liste d'instructions typées, nous générons l'ensemble exhaustif de chemins d'attaque, avec chacun des chemins des positions et des longueurs spécifiques pour chacune des fautes. Enfin, ces chemins d'attaque sont validés à l'aide de la méthode d'instrumentation décrite dans la section 4.2.2.

Configuration pour les campagnes d'injection

D'après les observations issues de nos expérimentations préliminaires, nous avons identifié plusieurs amplitudes pouvant induire une faute de type rejeu d'instructions. Nous avons restreint notre sélection à dix amplitudes que nous essayons dans un ordre décroissant : $amplitude \in \{538, 536, 534, 532, 526, 524, 522, 500, 495, 490\}$

Pour ces évaluations, nous souhaitons limiter la durée d'une campagne d'injection pour un chemin d'attaque donné à environ 24 heures. Le temps de programmation de la plateforme TRAITOR augmentant avec le nombre de fautes, elle peut effectuer au minimum deux attaques par seconde. Comme une heure compte 3600 secondes, cela équivaut à 1800 attaques par heure. Nous considérons dix amplitudes d'horloge à tester, ce qui implique, sans tenir compte des approximations liées aux délais et durées des fautes, que SAMPLAI peut tester jusqu'à 180 configurations de fautes par heure. Par conséquent, nous fixons le nombre total de configurations à tester à $180 \times 24 = 4320$. Les observations expérimentales ont montré un taux élevé de reproductibilité des fautes, donc nous fixons à deux le nombre de répétitions pour chaque configuration de fautes. Concernant la génération des intervalles des délais pour chacune faute, nous fixons à 3 la valeur de k, donc les intervalles sont définis tels que $délai_n \pm 3$.

4.3.2 Résultats expérimentaux

Nous commençons par présenter les chemins d'attaque identifiés par l'analyse de binaires. Ensuite, nous détaillons les résultats des campagnes d'injection de fautes, incluant plusieurs attaques réussies nécessitant jusqu'à huit fautes.

Version	Nbr. de chemins d'attaque	Nbr. de fautes		Nbr. d'instr. sautées	
		Min.	Max.	Min.	Max.
V0a	11	1	1	5	16
V0b	12	1	4	6	16
V1a	21	1	1	5	18
V1b	28	1	4	4	23
V2a	11	1	3	6	16
V2b	13	1	6	7	19
V3a	30	1	5	9	44
V3b	500	1	5	11	46
V4a	542	4	5	40	76
V4b	286	4	9	30	71
V5a	740	1	7	10	28
V5b	556	1	3	10	26
V6a	392	2	6	13	54
V6b	333	2	5	15	58
V7a	403	8	9	80	96
V7b	0				

FIGURE 4.9 – Résumé des chemins d'attaque identifiés par l'analyse

Chemins d'attaques identifiés

La figure 4.9 indique le nombre de chemins d'attaque identifiés par l'analyse de binaire, qui ont été validés, pour chaque version de `VerifyPIN`. Une version « Vpa » possède l'alignement de base et une version « Vpb » possède un décalage de 2 octets dans l'alignement. Bien que l'analyse ait réussi à trouver des chemins d'attaque pour toutes les versions, nous pouvons remarquer que l'alignement a un impact très significatif. Nous obtenons 30 chemins pour la version V3a, tandis que pour la version V3b, nous obtenons 500 chemins, soit plus de 16 fois plus. Cette différence est encore plus visible pour la version V7, où l'analyse trouve des chemins uniquement pour la version V7a. Le nombre de fautes à injecter et d'instructions à sauter varie significativement entre les versions de `VerifyPIN`. Les versions de V0 à V3 et la version V5 nécessitent qu'une seule faute et environ une dizaine d'instructions à sauter. En revanche, la version V4 requiert au moins quatre fautes, avec un nombre d'instructions à sauter allant de 30 à 76. Enfin, la version V7a requiert au moins huit fautes et un nombre d'instructions à sauter nettement plus élevé que pour les autres versions, compris entre 80 et 96 instructions.

Attaques physiques réalisées

Pour nos campagnes d'injections de fautes, nous sélectionnons les cinq chemins d'attaque ayant le plus petit nombre de fautes et les fautes les plus courtes pour chaque binaire attaqué. L'ensemble des campagnes d'injection est réalisé avec le même dispositif TRAITOR sur trois cartes du même modèle afin d'observer d'éventuelles variations dans les résultats. La figure 4.10 présente le nombre d'attaques ayant réussi pour chaque binaire attaqué selon l'appareil cible. Le taux de succès d'une attaque correspond au ratio d'attaques réussies sur le total des cinq attaques réalisées pour une version de `VerifyPIN` sur une cible donnée. Nous observons que, hormis la version V5b, toutes les versions de `VerifyPIN` sont vulnérables. Les taux de succès des attaques sont relativement similaires entre les cibles pour une version donnée. La cible 2 possède un taux de succès plus haut que les deux autres cibles pour les versions V0a, V1b, V2a et V3b. La version V5a est vulnérable pour les cibles 2 et 3, avec un taux de succès bas, car seulement un chemin d'attaque a abouti à une attaque concrète réussie.

Version	**Cible 1**		**Cible 2**		**Cible 3**	
	Nbr d'att. réussies	**Taux de succès**	**Nbr d'att. réussies**	**Taux de succès**	**Nbr d'att. réussies**	**Taux de succès**
V0a	1	0,2	2	0,4	1	0,2
V0b	5	1	5	1	5	1
V1a	3	0,6	3	0,6	3	0,6
V1b	4	0,8	2	0,4	4	0,8
V2a	1	0,2	2	0,4	1	0,2
V2b	4	0,8	5	1	5	1
V3a	5	1	5	1	5	1
V3b	2	0,4	4	0,8	2	0,4
V4a	5	1	5	1	5	1
V4b	5	1	5	1	5	1
V5a	0	0	1	0,2	1	0,2
V5b	0	0	0	0	0	0
V6a	5	1	5	1	5	1
V6b	5	1	5	1	5	1
V7a	4	0,8	4	0,8	1	0,2

FIGURE 4.10 – Résumé des attaques physiques réussies

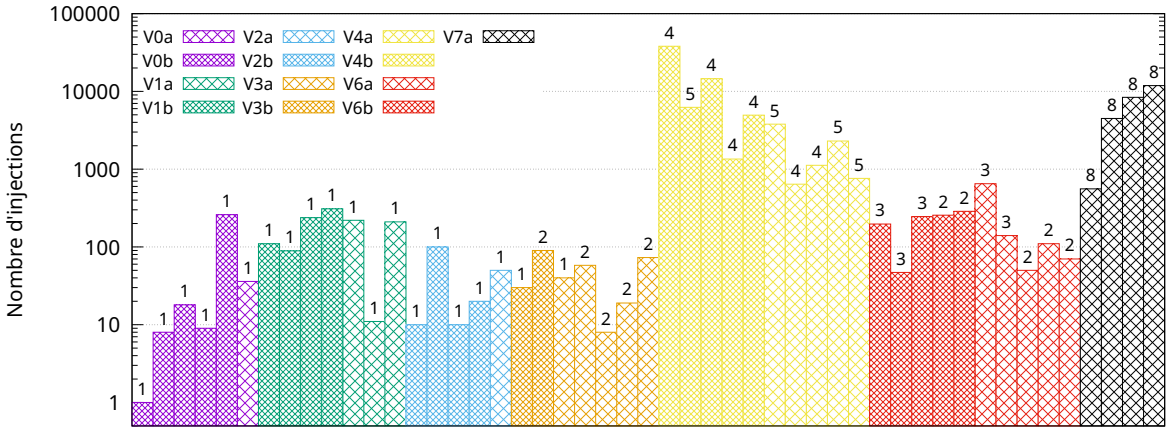
La figure 4.11 présente un histogramme pour chacune des cibles, avec une barre pour chacun des chemins d'attaque qui ont mené à une attaque réussie dont la taille correspond

aux nombres de tentatives requises. Les barres sont annotées avec le nombre de fautes injectés durant l'attaque. Nous observons que la majorité des attaques nécessitent moins de 1000 tentatives. Seulement les versions V4 sur la cible 1 et la version V7a sur toutes les cibles nécessitent davantage de tentatives, autour de 10000. Notre banc d'injection permet de réaliser 2 à 5 tentatives d'attaque par seconde, comprenant la communication avec la cible et l'injection de fautes. Cela signifie qu'un délai entre 10 minutes et 1 heure 30 minutes permet de mener une attaque réussie sur l'ensemble des binaires. L'heuristique de réduction du nombre de configurations peut influencer le nombre de tentatives nécessaires avant qu'une attaque ne réussisse. En effet, la stratégie de limitation consiste à mélanger aléatoirement la liste puis de la tronquer, ce qui permet de préserver une représentativité par échantillonnage de l'ensemble total des configurations tout en réduisant sa taille. Par conséquent, la sélection des configurations pour un même chemin peut varier d'une campagne à l'autre. Nous pouvons toutefois observer que le nombre de tentatives tend à augmenter proportionnellement avec le nombre et à la largeur des fautes requises par une attaque.

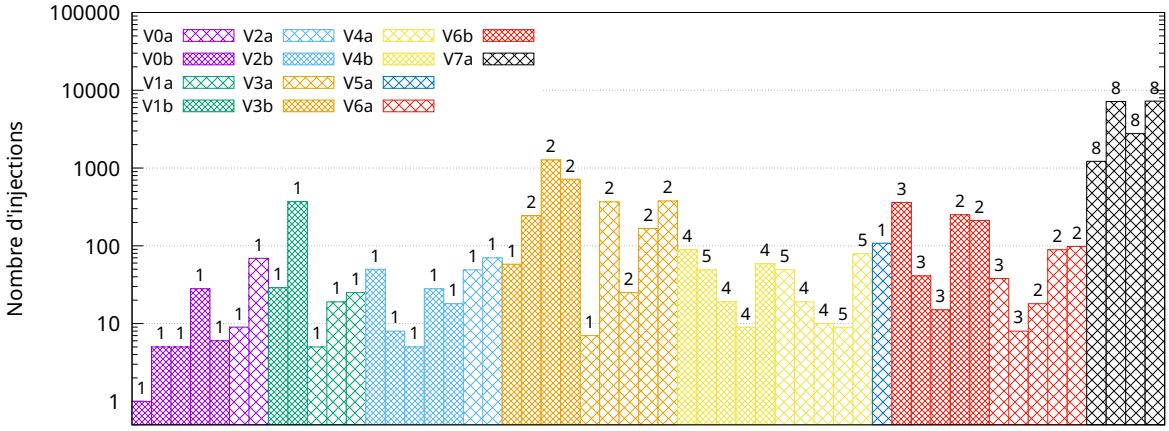
La figure 4.12 illustre graphiquement les fautes ayant conduit à des attaques réussies. Pour chaque version de `VerifyPIN`, elle regroupe l'ensemble des configurations de fautes ayant permis de réaliser une attaque réussie sur l'une des trois cibles. Chaque ligne correspond à une configuration de faute, où l'axe des abscisses indique le temps exprimé en nombre de cycles écoulés depuis la réception du signal de synchronisation. Les segments sur les graphiques indiquent le point de départ des fautes et leur durée. Par ailleurs, cette figure illustre aussi le nombre de fautes de chaque attaque réussie.

D'après les résultats obtenus, nous constatons des attaques par injection de fautes réussies pour toutes les versions de `VerifyPIN`, avec un taux de réussite généralement élevé, à l'exception des versions V5. Les résultats montrent des différences significatives entre les cartes cibles, notamment le nombre de tentatives nécessaires et dans une moindre mesure, le nombre d'attaques réussies. Trois facteurs peuvent expliquer ces écarts :

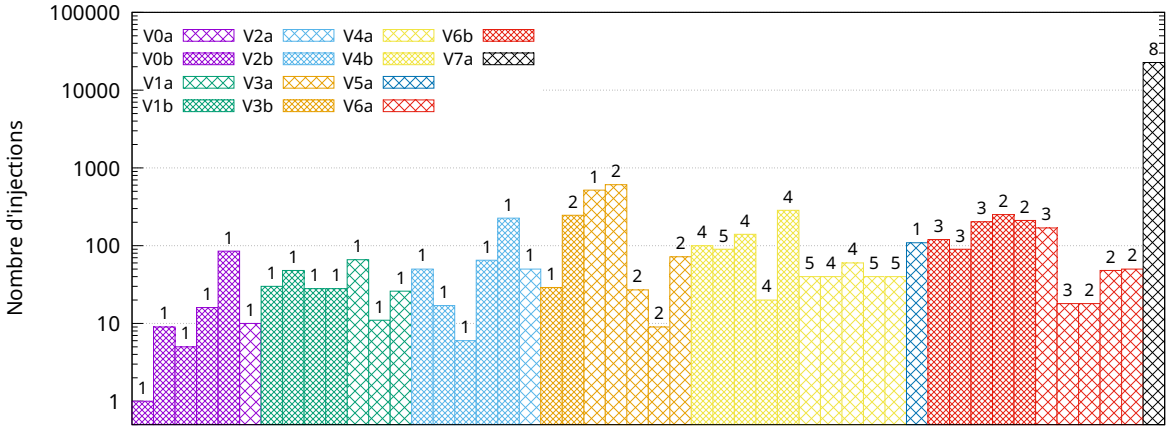
— Variabilité des effets de fautes : la reproductibilité des effets des fautes n'est pas systématique et plusieurs essais peuvent être nécessaires avant d'obtenir l'effet de rejeu souhaité. Cependant, notre stratégie d'exploration privilégie, selon l'imbrication des boucles décrite en section 4.2, la diversité des configurations aux répétitions. Ainsi, une configuration valide n'est répétée qu'après avoir exploré de nouveau l'ensemble des paramètres temporels. Pour des attaques impliquant de nombreuses fautes, comme pour les versions V4 et V7a, cette approche peut s'avérer coûteuse



(a) Résultats pour la cible 1

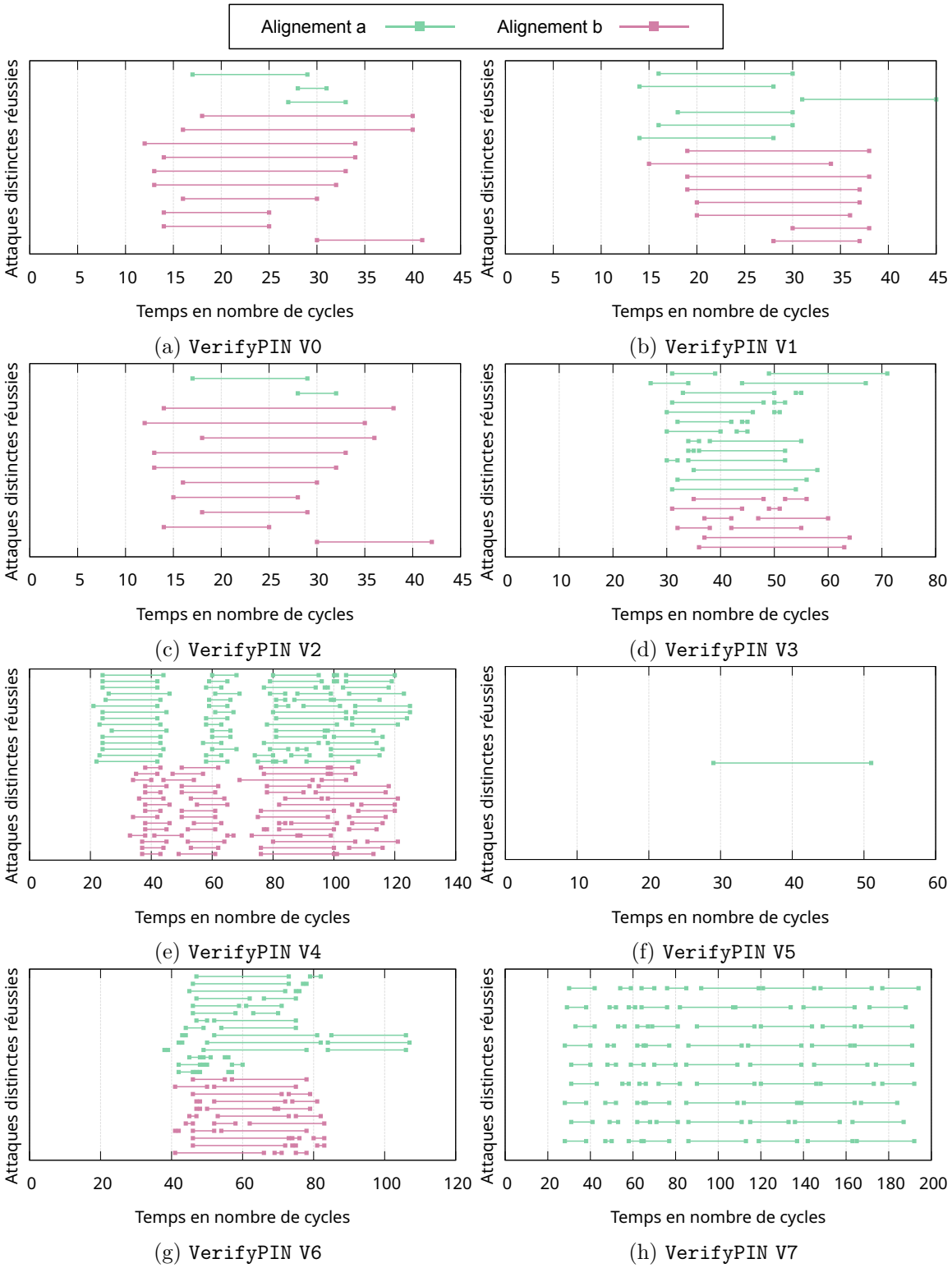


(b) Résultats pour la cible 2



(c) Résultats pour la cible 3

FIGURE 4.11 – Nombre d'injections réalisées avant la réussite d'une attaque (échelle logarithmique). Le nombre de fautes nécessaires pour chaque attaque est indiqué en haut des barres.



(a) `VerifyPIN V0` (b) `VerifyPIN V1`

(c) `VerifyPIN V2` (d) `VerifyPIN V3`

(e) `VerifyPIN V4` (f) `VerifyPIN V5`

(g) `VerifyPIN V6` (h) `VerifyPIN V7`

FIGURE 4.12 – Chemins d'attaque distincts ayant mené à une attaque réelle réussie sur une des cibles pour chaque version de `VerifyPIN`

en temps et conduire à des écarts notables entre les campagnes.

— Limitation de l'ensemble des configurations : le nombre de configurations à tester explose avec le nombre de fautes requises par l'attaque. Comme évoqué précédemment, nous utilisons une stratégie pour limiter le nombre de configurations, qui applique des mélanges aléatoires de la liste finale des configurations de fautes. Ce procédé rend cette liste potentiellement différente entre les campagnes pour un même chemin d'attaque, pouvant provoquer des variations dans les résultats.

— Variabilité matérielle : une dernière explication plus hypothétique serait que les cartes que nous attaquons sont fonctionnellement identiques, c'est-à-dire fournissent les mêmes spécifications techniques et mêmes fonctionnalités, mais présenterait des variations matérielles. Par exemple, des puces différentes peuvent être utilisées entre les séries d'un même modèle ou encore des défauts présents le silicium. Bien que ces différences soient minimes, cela pourrait impacter les effets des fautes. L'étape de caractérisation a été réalisée seulement sur une carte cible pour l'ensemble des cartes attaquées. Une étape de caractérisation indépendante à chaque carte pourrait limiter ce facteur matériel.

Les contraintes de l'analyse sur le positionnement et la taille des fautes découlent d'une campagne de caractérisation, qui s'appuie généralement sur un ensemble restreint de codes de test. Cela limite le nombre de combinaisons d'instructions ayant été testées. En conséquence, les contraintes identifiées peuvent être incomplètes, ce qui signifie que certains chemins d'attaque identifiés par l'analyse ne sont pas toujours réalisables en pratique, comme nous pouvons observer pour la version V5.

In fine, notre approche réussit à identifier et réaliser des attaques pour 14 des 16 programmes évalués. La figure 4.12 montre que parmi les 60 attaques réussies, 23 nécessitent une seule faute, 13 en nécessitent deux, 5 en nécessitent trois, 6 en nécessitent quatre, 4 en nécessitent cinq, et enfin, 9 en nécessitent huit. D'après le nombre minimal d'instructions devant être sautées pour chacune des versions de `VerifyPIN`, présenté dans la figure 4.10, la version V7a a nécessité des attaques impliquant le saut de plus de 80 instructions. Ces résultats démontrent la capacité de SAMPLAI à réaliser de manière automatique des attaques multi-fautes.

4.3.3 Suggestions de contre-mesures

De ces expérimentations, nous dérivons des suggestions à l'intention des concepteurs de contre-mesures. Celles-ci sont conceptuelles et ne visent qu'à protéger un programme contre les attaques par rejeu d'instructions et peuvent entraîner d'autres effets de bord. Des recherches complémentaires sont nécessaires pour confirmer leur efficacité.

Éviter les instructions idempotentes

Les fautes de type rejeu d'instructions sont réalisables lorsque les instructions rejouées sont idempotentes. Par conséquent, une ligne directrice évidente consiste à éviter autant que possible les instructions idempotentes. Cependant, de nombreuses contre-mesures logicielles impliquent la duplication des instructions afin de résister aux sauts d'instructions ou à la modification des données en mémoire [72], [110]. Le schéma de duplication augmente généralement le nombre d'instructions idempotentes dans le binaire [111]. Par conséquent, ces contre-mesures peuvent augmenter la surface d'attaque. Pour préserver les avantages de ces contre-mesures, les instructions doivent être désalignées, comme indiqué ci-dessous.

Désaligner délibérément les instructions

Pour éviter un plantage, la faute doit être injectée après l'exécution d'un mot de mémoire aligné. Une contre-mesure efficace et relativement peu coûteuse consiste à désaligner délibérément les instructions qui entourent les séquences d'instructions les plus critiques. Le désalignement peut être obtenu en insérant des instructions de 32 ou 16 bits sans effet (telles que `NOP` ou `EOR.W R0,R0,R0`) de sorte que les mots mémoire contiennent toujours la moitié d'une instruction de 32 bits. Cela garantit que les mots mémoire ne peuvent pas être rejoués. Cependant, afin de ne pas créer de nouvelles vulnérabilités, les instructions insérées doivent être soigneusement choisies pour ne pas offrir la possibilité de créer une instruction de 16 bits à partir d'une partie d'une instruction de 32 bits mal alignée, comme l'ont montré Alshaer et al. [56].

4.4 Conclusion

Nous proposons SAMPLAI, une approche complète pour mener des attaques par injections multi-fautes. Elle comprend une analyse de programme binaire basée sur SAMVA,

supportant initialement les attaques par saut d'instructions sans prendre en considération les contraintes liées à la réalisation pratique des effets de la faute. Dans cette extension, nous intégrons le modèle de rejeu des instructions afin d'obtenir des chemins d'attaque intégrant cet effet observé des fautes. SAMPLAI comprend également une méthode pour convertir les chemins d'attaque identifiés en paramètres temporels d'injection de fautes et automatise les campagnes d'attaque.

Nous évaluons notre approche en utilisant la plateforme d'horloge TRAITOR et une carte embarquée contenant un processeur Cortex-M3. Nous ciblons huit versions de programmes de vérification de codes PIN avec des contre-mesures logicielles pour les attaques par injection de fautes. Les résultats montrent que SAMPLAI attaque avec succès douze versions, même lorsque l'attaque nécessite d'injecter jusqu'à huit fautes. De plus, le temps de réussite peut être maintenu acceptable grâce à l'exploration des configurations des fautes.

Chapitre 5

Conclusion

Au cours de cette thèse, nous avons étudié les méthodes d'analyse de programme permettant d'identifier les vulnérabilités potentielles résultant d'attaques par injection de fautes. Notre objectif principal était de développer une méthode permettant de passer à l'échelle dans un contexte où un attaquant peut injecter de nombreuses fautes. Pour répondre à cet objectif, nous avons proposé SAMVA, une approche d'analyse de code binaire capable de construire des chemins d'attaque pour des fautes multiples causant des sauts de plusieurs instructions. Nous avons ensuite conçu et développé SAMPLAI, une méthode permettant d'automatiser l'exécution physique des attaques par injection de fautes en exploitant les chemins d'attaque identifiés par SAMVA. Les principales contributions de cette thèse sont résumées dans la section 5.1, tandis que la section 5.2 décrit les potentielles perspectives de recherche.

5.1 Contributions clefs

Nos principales contributions sont résumées ci-dessous.

Analyse statique de détermination de chemins d'attaque multi-fautes

Nous proposons une solution permettant d'évaluer les vulnérabilités d'un programme binaire face à des attaques impliquant des sauts d'instructions multiples. Afin de gérer l'explosion combinatoire qui survient lorsque l'on considère tous les effets de fautes possibles avec de nombreux paramètres, notre approche se base uniquement sur de l'analyse statique et des stratégies d'exploration des chemins. Notre outil nommé SAMVA implémente cette analyse. Cette implémentation est développée en langage Python et C++ et comprend environ 9.7k lignes de code. SAMVA est conçu pour être modulaire et extensible, afin de faciliter l'ajout de nouvelles contraintes supplémentaires concernant les positions ou les tailles des fautes. Nous avons évalué SAMVA en identifiant des chemins

d'attaque sur un ensemble de programmes qui intègrent diverses contre-mesures contre les fautes. Les résultats obtenus montent que notre solution est capable de détecter des chemins nécessitant de nombreuses fautes, même dans les programmes les plus protégés et dans un temps très réduit. Ces résultats démontrent que l'utilisation d'heuristiques permet de passer à l'échelle avec le nombre de fautes. Ce type d'analyse semble prometteur et offre encore de nombreuses possibilités d'amélioration.

Automatisation de la réalisation des attaques multi-fautes

Après avoir développé notre outil de recherche de vulnérabilités, nous avons cherché à compléter l'ensemble des étapes composant une campagne d'injection de fautes. À ce titre, nous proposons SAMPLAI, une méthode complète qui réunit la recherche de chemins d'attaque et la réalisation des injections sur une cible matérielle concrète. Notre approche comprend : 1) une extension de SAMVA permettant de prendre en compte les conditions de réalisation des fautes pour des instructions ayant pour effet le rejeu d'instruction ; 2) une méthode de conversion des chemins d'attaque, qui spécifient seulement quelles sont les instructions devant être fautées, en un ensemble de paramètres temporels ; 3) une automatisation des attaques incluant l'exploration des paramètres temporels et l'intensité de l'injection, le paramétrage de la plateforme d'injection, la communication avec la cible et le traitement des résultats obtenus. Nous avons évalué cette méthode en utilisant les mêmes programmes que pour l'évaluation de SAMVA, en utilisant la plateforme d'injection TRAITOR et trois cartes embarquées équipées d'un microprocesseur Cortex-M3. Notre méthode a permis d'attaquer avec succès la majorité des programmes, malgré les contre-mesures, avec des attaques contenant jusqu'à huit fautes. Suite à nos observations, nous donnons quelques suggestions pour concevoir de meilleures contre-mesures contre ce type d'attaque.

5.2 Perspectives

Nous avons présenté une analyse de programme pour identifier des chemins d'attaque et une méthode pour convertir ces chemins en paramètres physiques pour les moyens d'injection de fautes et automatiser la réalisation des attaques. Pour le futur, nous formulons quelques suggestions afin d'améliorer ces deux approches.

5.2.1 Amélioration de l'analyse

Intégrer de l'analyse de données

Pour des raisons de performance, SAMVA se concentre uniquement sur l'analyse du flot de contrôle résultant de l'exécution ou du saut des instructions de branchement, sans considérer les aspects relatifs aux données. Cette limitation est forte, car en conséquence, tous les branchements conditionnels sont systématiquement fautés, et ce, même lorsque la condition du saut est satisfaite. De plus, l'algorithme de positionnement des fautes employé peut trouver des solutions qui mènent à des flots de contrôle incohérents. Un plantage peut alors intervenir au cours de l'exécution des chemins d'attaque, notamment à cause d'accès mémoire illégaux. Une étape de validation des chemins par simulation ou exécution pas à pas est utilisée pour réduire les chemins incorrects. Intégrer de l'analyse de données dans un contexte multi-faute peut permettre de réduire le nombre de fautes requis et les chemins trouvés seraient alors moins susceptibles de causer un plantage. Néanmoins, une telle analyse est difficile à concevoir dans la mesure où les états des données possibles augmentent aussi de manière exponentielle avec le nombre de fautes. Par conséquent, des heuristiques suffisantes doivent aussi être imaginées pour répondre à ces problématiques. Par exemple, une analyse des données du registre d'état, qui détermine si la condition d'un saut est remplie, ou du registre de lien, utilisé pour le retour de fonction, peut permettre d'ajouter de nouveaux flots de contrôle possibles dans SAMVA.

Considérer davantage d'effets de faute

Lors nos travaux avec SAMPLAI, nous nous sommes intéressés au rejeu d'instructions, une forme de saut d'instructions plus représentative avec ce qui est possible de réaliser avec la plateforme TRAITOR. Il serait envisageable de tirer parti de l'extensibilité de SAMVA pour ajouter de nouveaux types de faute. Les bancs d'injection laser, par exemple, permettent de manipuler les valeurs des bits en mémoire (*bit-set*). De ce fait, il est possible de modifier des instructions de telle sorte qu'elles n'aient plus d'effet sur l'exécution du programme, afin de simuler l'effet d'un saut d'instruction. Ajouter de telles contraintes dans SAMVA permettrait d'identifier des chemins d'attaque compatibles avec davantage de moyens d'injection, tout en conservant la modélisation du programme sous forme de CFG.

Permettre de nouveaux types d'exploit

SAMVA permet d'identifier des chemins pour des attaques d'accessibilité, où le flot de contrôle du programme est détourné afin d'atteindre des sections spécifiques ciblées par un attaquant. En complément à l'éventuel ajout d'une analyse de données, il serait possible d'étendre le modèle d'accessibilité en intégrant la notion d'état des registres et de la mémoire à l'entrée des sections visées.

Supporter de nouvelles architectures

L'implémentation de notre analyse est basée sur le projet *angr* [87], qui permet d'extraire une représentation intermédiaire du programme et construire un CFG. Cette représentation constitue une abstraction commune à toutes les architectures prises en charge par *angr*. L'essentiel de l'implémentation de SAMVA utilise cette abstraction, rendant ainsi le code déjà compatible avec les architectures supportées. Cependant, certains aspects de l'analyse dépendent de spécificités de l'architecture, comme la taille des instructions ou les noms des registres. Actuellement, ces éléments sont programmés exclusivement pour Arm et nécessiteraient d'être portés pour d'autres architectures. SAMVA est néanmoins contraint par les architectures supportées par *angr*. À ce jour, *angr* prend en charge les architectures Arm, x86, MIPS, et PowerPC, avec un support expérimental pour RISC-V.

5.2.2 Faciliter la réalisation des fautes

Calibration automatique

La calibration est une étape nécessaire et préalable à une campagne d'injection de fautes. Celle-ci consiste à déterminer les paramètres d'injection spécifiques à la plateforme d'injection. Des paramètres sont constants, tels que le délai potentiel du mécanisme de synchronisation, et d'autres sont variables en fonction d'appareil cible, comme les plages d'intensité des perturbations permettant d'introduire des fautes. Toutefois, les résultats obtenus lors de l'évaluation de SAMPLAI affichent des différences significatives entre les trois cartes attaquées. Par conséquent, réaliser une calibration distincte pour chaque cible permettrait d'affiner les paramètres d'injection et, à terme, optimiserait l'efficacité des attaques. Des travaux sur la calibration automatique des plateformes d'injection existent [60] et peuvent être intégrés à SAMPLAI.

Automatiser la caractérisation des fautes

Dans le cadre des travaux pour SAMPLAI, nous avons automatisé l'acquisition des données nécessaires à la caractérisation des fautes. Cependant, la caractérisation en soit reste un processus manuel et fastidieux, en raison du grand nombre de codes de test à évaluer et de la grande quantité de données issues des injections. En réutilisant des éléments de l'infrastructure mise en place, qui intègre l'analyse de binaire et l'instrumentation des cartes cibles, il serait envisageable d'automatiser, au moins partiellement, cette étape chronophage. Cela permettrait de mieux connaitre les capacités de la plateforme d'injection sur la cible attaquée et d'extraire avec plus de précision les conditions de réalisation des fautes. En outre, le gain de temps ainsi obtenu permettrait d'utiliser un plus grand nombre de codes de test.

BIBLIOGRAPHIE

[1] D. Boneh, R. A. DeMillo et R. J. Lipton, « On the Importance of Checking Cryptographic Protocols for Faults », in *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, W. Fumy, éd., sér. Lecture Notes in Computer Science, t. 1233, Springer, 1997, p. 37-51. doi : 10.1007/3-540-69053-0_4.

[2] M. Witteman et M. Oostdijk, *Secure application programming in the presence of side channel attacks*, 2008. adresse : https://www.riscure.com/publication/secure-application-programming-presence-side-channel-attacks/.

[3] K. Heydemann, J.-F. Lalande et P. Berthomé, « Formally verified software countermeasures for control-flow integrity of smart card C code », *Comput. Secur.*, t. 85, p. 202-224, 2019. doi : 10.1016/J.COSE.2019.05.004.

[4] B. Colombier, P. Grandamme, J. Vernay et al., « Multi-Spot Laser Fault Injection Setup : New Possibilities for Fault Injection Attacks », in *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, V. Grosso et T. Pöppelmann, éd., sér. Lecture Notes in Computer Science, t. 13173, Springer, 2021, p. 151-166. doi : 10.1007/978-3-030-97348-3_9.

[5] L. Rivière, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer et 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 2015, Washington, DC, USA, 5-7 May, 2015*, IEEE Computer Society, 2015, p. 62-67. doi : 10.1109/HST.2015.7140238.

[6] S. K. Bukasa, R. Lashermes, J.-L. Lanet et A. Legay, « Let's shock our IoT's heart : ARMv7-M under (fault) attacks », in *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, S. Doerr, M. Fischer, S. Schrittwieser et D. Herrmann, éd., ACM, 2018, 33 :1-33 :6. doi : 10.1145/3230833.3230842.

[7] J.-M. Dutertre, T. Riom, O. Potin et J.-B. Rigaud, « Experimental Analysis of the Laser-Induced Instruction Skip Fault Model », in *Secure IT Systems - 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18-20, 2019, Proceedings*, A. Askarov, R. R. Hansen et W. Rafnsson, éd., sér. Lecture Notes in Computer Science, t. 11875, Springer, 2019, p. 221-237. doi : 10.1007/978-3-030-35055-0_14.

[8] A. Menu, J.-M. Dutertre, O. Potin, J.-B. Rigaud et J.-L. Danger, « Experimental Analysis of the Electromagnetic Instruction Skip Fault Model », in *15th Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2020, Marrakech, Morocco, April 1-3, 2020*, IEEE, 2020, p. 1-7. doi : 10.1109/DTIS48698.2020.9081261.

[9] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud et J.-L. Danger, « Laser-induced Single-bit Faults in Flash Memory : Instructions Corruption on a 32-bit Microcontroller », in *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*, IEEE, 2019, p. 1-10. doi : 10.1109/HST.2019.8741030.

[10] L. Claudepierre, P.-Y. Péneau, D. Hardy et E. Rohou, « TRAITOR : A Low-Cost Evaluation Platform for Multifault Injection », in *ASSS '21 : Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems, Virtual Event, Hong Kong, 7 June, 2021*, W. Meng et L. Li, éd., ACM, 2021, p. 51-56. doi : 10.1145/3457340.3458303.

[11] *Common Criteria Portal*. adresse : https://www.commoncriteriaportal.org.

[12] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas et J. Clédière, « From Code Review to Fault Injection Attacks : Filling the Gap Using Fault Model Inference », in *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, N. Homma et M. Medwed, éd., sér. Lecture Notes in Computer Science, t. 9514, Springer, 2015, p. 107-124. doi : 10.1007/978-3-319-31271-2_7.

[13] V. Werner, L. Maingault et M.-L. Potet, « An End-to-End Approach for Multi-Fault Attack Vulnerability Assessment », in *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, IEEE, 2020, p. 10-17. doi : 10.1109/FDTC51366.2020.00009.

[14] M.-L. Potet, L. Mounier, M. Puys et L. Dureuil, « Lazart : A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections », in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, IEEE Computer Society, 2014, p. 213-222. doi : 10.1109/ICST.2014.34.

[15] S. Ducousso, S. Bardin et M.-L. Potet, « Adversarial Reachability for Program-level Security Analysis », in *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, T. Wies, éd., sér. Lecture Notes in Computer Science, t. 13990, Springer, 2023, p. 59-89. doi : 10.1007/978-3-031-30044-8_3.

[16] J.-B. Bréjon, K. Heydemann, E. Encrenaz, Q. Meunier et S.-T. Vu, « Fault attack vulnerability assessment of binary code », in *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, sér. CS2 '19, Valencia, Spain : Association for Computing Machinery, 2019, p. 13-18, isbn : 9781450361828. doi : 10.1145/3304080.3304083.

[17] S. Tollec, M. Asavoae, D. Couroussé, K. Heydemann et M. Jan, « Exploration of Fault Effects on Formal RISC-V Microarchitecture Models », in *Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2022, Virtual Event / Italy, September 16, 2022*, IEEE, 2022, p. 73-83. doi : 10.1109/FDTC57191.2022.00017.

[18] S. Tollec, M. Asavoae, D. Couroussé, K. Heydemann et M. Jan, « μARCHIFI : Formal Modeling and Verification Strategies for Microarchitectural Fault Injections », in *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, A. Nadel et K. Y. Rozier, éd., IEEE, 2023, p. 101-109. doi : 10.34727/2023/ISBN.978-3-85448-060-0_18.

[19] A. Gicquel, D. Hardy, K. Heydemann et E. Rohou, « SAMVA : Static Analysis for Multi-fault Attack Paths Determination », in *Constructive Side-Channel Analysis and Secure Design - 14th International Workshop, COSADE 2023, Munich, Germany, April 3-4, 2023, Proceedings*, E. B. Kavun et M. Pehl, éd., sér. Lecture Notes in Computer Science, t. 13979, Springer, 2023, p. 3-22. doi : 10.1007/978-3-031-29497-6_1.

[20] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen et P. de Choudens, « FISSC : A Fault Injection and Simulation Secure Collection », in *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, A. Skavhaug, J. Guiochet et F. Bitsch, éd., sér. Lecture Notes in Computer Science, t. 9922, Springer, 2016, p. 3-11. doi : 10.1007/978-3-319-45477-1_1.

[21] S. Morgan, *Cybercrime To Cost The World $10.5 Trillion Annually By 2025*, 2020. adresse : https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/.

[22] Z. G. Ruthberg et R. G. McKenzie, « Audit and evaluation of computer security », 1977.

[23] *La directive NIS 2*, 2025. adresse : https://cyber.gouv.fr/la-directive-nis-2.

[24] G. Balakrishnan et T. W. Reps, « WYSINWYX : What you see is not what you eXecute », *ACM Trans. Program. Lang. Syst.*, t. 32, *6*, 23 :1-23 :84, 2010. doi : 10.1145/1749608.1749612.

[25] Z. Durumeric, J. Kasten, D. Adrian et al., « The Matter of Heartbleed », in *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, C. Williamson, A. Akella et N. Taft, éd., ACM, 2014, p. 475-488. doi : 10.1145/2663716.2663755.

[26] H. Shacham, « The geometry of innocent flesh on the bone : return-into-libc without function calls (on the x86) », in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati et P. F. Syverson, éd., ACM, 2007, p. 552-561. doi : 10.1145/1315245.1315313.

[27] P. Kocher, J. Horn, A. Fogh et al., « Spectre attacks : exploiting speculative execution », *Commun. ACM*, t. 63, *7*, p. 93-101, 2020. doi : 10.1145/3399742.

[28] P. C. Kocher, J. Jaffe et B. Jun, « Differential Power Analysis », in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, M. J. Wiener, éd., sér. Lecture Notes in Computer Science, t. 1666, Springer, 1999, p. 388-397. doi : 10.1007/3-540-48405-1_25.

[29] M. G. Kuhn, « Compromising emanations : eavesdropping risks of computer displays », Thèse de doctorat, University of Cambridge, UK, 2002.

[30] E. Biham et A. Shamir, « Differential Fault Analysis of Secret Key Cryptosystems », in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, B. S. K. Jr., éd., sér. Lecture Notes in Computer Science, t. 1294, Springer, 1997, p. 513-525. doi : 10.1007/BFB0052259.

[31] G. Piret et J.-J. Quisquater, « A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD », in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, C. D. Walter, Ç. K. Koç et C. Paar, éd., sér. Lecture Notes in Computer Science, t. 2779, Springer, 2003, p. 77-88. doi : 10.1007/978-3-540-45238-6_7.

[32] S.-M. Yen et M. Joye, « Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis », *IEEE Trans. Computers*, t. 49, *9*, p. 967-970, 2000. doi : 10.1109/12.869328.

[33] S. P. Skorobogatov et R. J. Anderson, « Optical Fault Induction Attacks », in *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, B. S. K. Jr., Ç. K. Koç et C. Paar, éd., sér. Lecture Notes in Computer Science, t. 2523, Springer, 2002, p. 2-12. doi : 10.1007/3-540-36400-5_2.

[34] J. Breier, D. Jap et C.-N. 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, CPSS 2015, Singapore, Republic of Singapore, April 14 - March 14, 2015*, J. Zhou et D. Jones, éd., ACM, 2015, p. 99-103. doi : 10.1145/2732198.2732206.

[35] E. Trichina et R. Korkikyan, « Multi Fault Laser Attacks on Protected CRT-RSA », in *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, USA, 21 August 2010*, L. Breveglieri, M. Joye, I. Koren, D. Naccache et I. Verbauwhede, éd., IEEE Computer Society, 2010, p. 75-86. doi : 10.1109/FDTC.2010.14.

[36] V. Khuat, J.-M. Dutertre et J.-L. Danger, « Software countermeasures against the multiple instructions skip fault model », *Microelectronics Reliability*, t. 155, p. 115 370, 2024.

[37] T. Trouchkine, S. K. Bukasa, M. Escouteloup, R. Lashermes et G. Bouffard, « Electromagnetic fault injection against a System-on-Chip, toward new micro-architectural fault models », *CoRR*, t. abs/1910.11566, 2019. arXiv : 1910.11566. adresse : http://arxiv.org/abs/1910.11566.

[38] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson et E. Encrenaz, « Electromagnetic Fault Injection : Towards a Fault Model on a 32-bit Microcontroller », in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, W. Fischer et J.-M. Schmidt, éd., IEEE Computer Society, 2013, p. 77-88. doi : 10.1109/FDTC.2013.9.

[39] J. Proy, K. Heydemann, F. Majéric, A. Cohen et A. Berzati, « Studying EM Pulse Effects on Superscalar Microarchitectures at ISA Level », *CoRR*, t. abs/1903.02623, 2019. arXiv : 1903.02623. adresse : http://arxiv.org/abs/1903.02623.

[40] M. Dumont, M. Lisart et P. Maurine, « Electromagnetic Fault Injection : How Faults Occur », in *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*, IEEE, 2019, p. 9-16. doi : 10.1109/FDTC.2019.00010.

[41] J. Balasch, B. Gierlichs et I. Verbauwhede, « An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs », in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, L. Breveglieri, S. Guilley, I. Koren, D. Naccache et J. Takahashi, éd., IEEE Computer Society, 2011, p. 105-114. doi : 10.1109/FDTC.2011.9.

[42] Z. Kazemi, A. Papadimitriou, I. Souvatzoglou et al., « On a Low Cost Fault Injection Framework for Security Assessment of Cyber-Physical Systems : Clock Glitch Attacks », in *4th IEEE International Verification and Security Workshop, IVSW 2019, Rhodes Island, Greece, July 1-3, 2019*, IEEE, 2019, p. 7-12. doi : 10.1109/IVSW.2019.8854391.

[43] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle et P. Maistri, « Microarchitectural Insights into Unexplained Behaviors Under Clock Glitch Fault Injection », in *Smart Card Research and Advanced Applications - 22nd International Conference, CARDIS 2023, Amsterdam, The Netherlands, November 14-16, 2023, Revised Selected Papers*, S. Bhasin et T. Roche, éd., sér. Lecture Notes in Computer Science, t. 14530, Springer, 2023, p. 3-22. doi : 10.1007/978-3-031-54409-5_1.

[44] A. Marotta, R. Lashermes, G. Bouffard, O. Sentieys et R. Dafali, « Characterizing and Modeling Synchronous Clock-Glitch Fault Injection », in *Constructive Side-Channel Analysis and Secure Design - 15th International Workshop, COSADE 2024, Gardanne, France, April 9-10, 2024, Proceedings*, R. Wacquez et N. Homma, éd., sér. Lecture Notes in Computer Science, t. 14595, Springer, 2024, p. 3-21. doi : 10.1007/978-3-031-57543-3_1.

[45] A. Barenghi, G. Bertoni, E. Parrinello et G. Pelosi, « Low Voltage Fault Attacks on the RSA Cryptosystem », in *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*, L. Breveglieri, I. Koren, D. Naccache, E. Oswald et J.-P. Seifert, éd., IEEE Computer Society, 2009, p. 23-31. doi : 10.1109/FDTC.2009.30.

[46] C. Bozzato, R. Focardi et F. Palmarini, « Shaping the Glitch : Optimizing Voltage Fault Injection Attacks », *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, t. 2019, *2*, p. 199-224, 2019. doi : 10.13154/TCHES.V2019.I2.199-224.

[47] N. Timmers, A. Spruyt et M. Witteman, « Controlling PC on ARM Using Fault Injection », in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, IEEE Computer Society, 2016, p. 25-35. doi : 10.1109/FDTC.2016.18.

[48] Y. Kim, R. Daly, J. S. Kim et al., « Flipping bits in memory without accessing them : An experimental study of DRAM disturbance errors », in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, IEEE Computer Society, 2014, p. 361-372. doi : 10.1109/ISCA.2014.6853210.

[49] A. Tang, S. Sethumadhavan et S. J. Stolfo, « CLKSCREW : Exposing the Perils of Security-Oblivious Energy Management », in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda et T. Ristenpart, éd., USENIX Association, 2017, p. 1057-1074. adresse : https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[50] P. Qiu, D. Wang, Y. Lyu et G. Qu, « VoltJockey : Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies », in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang et J. Katz, éd., ACM, 2019, p. 195-209. doi : 10.1145/3319535.3354201.

[51] P. Grandamme, P.-A. Tissot, L. Bossuet, J.-M. Dutertre, B. Colombier et V. Grosso, « Switching Off your Device Does Not Protect Against Fault Attacks », *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, t. 2024, *4*, p. 425-450, 2024. doi : 10.46586/TCHES.V2024.I4.425-450.

[52] B. Yuce, P. Schaumont et M. Witteman, « Fault Attacks on Secure Embedded Software : Threats, Design, and Evaluation », *J. Hardw. Syst. Secur.*, t. 2, *2*, p. 111-130, 2018. doi : 10.1007/S41635-018-0038-1.

[53] D. S. V. Kumar, A. Beckers, J. Balasch, B. Gierlichs et I. Verbauwhede, « An In-Depth and Black-Box Characterization of the Effects of Laser Pulses on ATmega328P », in *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*, B. Bilgin et J.-B. Fischer, éd., sér. Lecture Notes in Computer Science, t. 11389, Springer, 2018, p. 156-170. doi : 10.1007/978-3-030-15462-2_11.

[54] M. S. Kelly, K. Mayes et J. F. Walker, « Characterising a CPU fault attack model via run-time data analysis », in *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*, IEEE Computer Society, 2017, p. 79-84. doi : 10.1109/HST.2017.7951802.

[55] T. Trouchkine, G. Bouffard et J. Clédière, « Fault Injection Characterization on Modern CPUs », in *Information Security Theory and Practice - 13th IFIP WG 11.2 International Conference, WISTP 2019, Paris, France, December 11-12,*

2019, Proceedings, M. Laurent et T. Giannetsos, éd., sér. Lecture Notes in Computer Science, t. 12024, Springer, 2019, p. 123-138. doi : 10.1007/978-3-030-41702-4_8.

[56] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle et P. Maistri, « Variable-Length Instruction Set : Feature or Bug ? », in *25th Euromicro Conference on Digital System Design, DSD 2022, Maspalomas, Spain, August 31 - Sept. 2, 2022*, IEEE, 2022, p. 464-471. doi : 10.1109/DSD57027.2022.00068.

[57] S. Nashimoto, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji et T. Aoki, « Buffer overflow attack with multiple fault injection and a proven countermeasure », *J. Cryptogr. Eng.*, t. 7, *1*, p. 35-46, 2017. doi : 10.1007/S13389-016-0136-3.

[58] J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, A. Tria et T. Vaschalde, « Fault Round Modification Analysis of the advanced encryption standard », in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2012, San Francisco, CA, USA, June 3-4, 2012*, IEEE Computer Society, 2012, p. 140-145. doi : 10.1109/HST.2012.6224334.

[59] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande, « High Level Model of Control Flow Attacks for Smart Card Functional Security », in *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20-24, 2012*, IEEE Computer Society, 2012, p. 224-229. doi : 10.1109/ARES.2012.79.

[60] V. Werner, L. Maingault et M.-L. Potet, « Fast Calibration of Fault Injection Equipment with Hyperparameter Optimization Techniques », in *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, V. Grosso et T. Pöppelmann, éd., sér. Lecture Notes in Computer Science, t. 13173, Springer, 2021, p. 121-138. doi : 10.1007/978-3-030-97348-3_7.

[61] O. Bittner, T. Krachenfels, A. Galauner et J.-P. Seifert, « The Forgotten Threat of Voltage Glitching : A Case Study on Nvidia Tegra X2 SoCs », in *18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021*, IEEE, 2021, p. 86-97. doi : 10.1109/FDTC53659.2021.00021.

[62] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu et D. Boneh, « On the effectiveness of address-space randomization », in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, V. Atluri, B. Pfitzmann et P. D. McDaniel, éd., ACM, 2004, p. 298-307. doi : 10.1145/1030083.1030124.

[63] A. Galauner, *Glitching the switch*, 2018. adresse : https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch.

[64] M. K. Temkin, *"Fusee gelee" exploit*, 2018. adresse : https://misc.ktemkin.com/fusee_gelee_nvidia.pdf.

[65] I. Verbauwhede, D. Karaklajic et J.-M. Schmidt, « The Fault Attack Jungle - A Classification Model to Guide You », in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, L. Breveglieri, S. Guilley, I. Koren, D. Naccache et J. Takahashi, éd., IEEE Computer Society, 2011, p. 3-8. doi : 10.1109/FDTC.2011.13.

[66] « Page GitHub du projet ibex », adresse : https://github.com/lowRISC/ibex.

[67] T. Chamelot, D. Couroussé et K. Heydemann, « SCI-FI : Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks », in *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*, C. Bolchini, I. Verbauwhede et I. Vatajelu, éd., IEEE, 2022, p. 556-559. doi : 10.23919/DATE54114.2022.9774685.

[68] T. Chamelot, D. Couroussé et K. Heydemann, « MAFIA : Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks », *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, t. 42, *12*, p. 4555-4568, 2023. doi : 10.1109/TCAD.2023.3276507.

[69] M. Joye et S.-M. Yen, « The Montgomery Powering Ladder », in *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, B. S. K. Jr., Ç. K. Koç et C. Paar, éd., sér. Lecture Notes in Computer Science, t. 2523, Springer, 2002, p. 291-302. doi : 10.1007/3-540-36400-5_22.

[70] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan et D. I. August, « SWIFT : Software Implemented Fault Tolerance », in *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005,*

San Jose, CA, USA, IEEE Computer Society, 2005, p. 243-254. DOI : 10.1109/CGO.2005.34.

[71] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi et F. Regazzoni, « Countermeasures against fault attacks on software implemented AES : effectiveness and cost », in *Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010*, ACM, 2010, p. 7. DOI : 10.1145/1873548.1873555.

[72] N. Belleville, K. Heydemann, D. Couroussé et al., « Automatic Application of Software Countermeasures Against Physical Attacks », in *Cyber-Physical Systems Security*, Ç. K. Koç, éd., Springer, 2018, p. 135-155. DOI : 10.1007/978-3-319-98935-8_7.

[73] J.-F. Lalande, K. Heydemann et P. Berthomé, « Software Countermeasures for Control Flow Integrity of Smart Card C Codes », in *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, M. Kutylowski et J. Vaidya, éd., sér. Lecture Notes in Computer Science, t. 8713, Springer, 2014, p. 200-218. DOI : 10.1007/978-3-319-11212-1_12.

[74] N. Theissing, D. Merli, M. Smola, F. Stumpf et G. Sigl, « Comprehensive analysis of software countermeasures against fault attacks », in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, E. Macii, éd., EDA Consortium San Jose, CA, USA / ACM DL, 2013, p. 404-409. DOI : 10.7873/DATE.2013.092.

[75] E. Boespflug, C. Ene, L. Mounier et M.-L. Potet, « Countermeasures Optimization in Multiple Fault-Injection Context », in *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, IEEE, 2020, p. 26-34. DOI : 10.1109/FDTC51366.2020.00011.

[76] E. Boespflug, L. Mounier, M.-L. Potet et A. Bouguern, « A compositional methodology to harden programs against multi-fault attacks », in *FDTC 2023 : Workshop on Fault Diagnosis and Tolerance in Cryptography 2023*, 2023.

[77] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick et P. Schaumont, « Software Fault Resistance is Futile : Effective Single-Glitch Attacks », in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography,*

FDTC 2016, Santa Barbara, CA, USA, August 16, 2016, IEEE Computer Society, 2016, p. 47-58. DOI : 10.1109/FDTC.2016.21.

[78] P.-Y. Péneau, L. Claudepierre, D. Hardy et E. Rohou, « NOP-Oriented Programming : Should we Care ? », in *IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2020, Genoa, Italy, September 7-11, 2020*, IEEE, 2020, p. 694-703. DOI : 10.1109/EUROSPW51379.2020.00100.

[79] E. M. Clarke, O. Grumberg et D. E. Long, « Model checking », in *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, M. Broy, éd., 1996, p. 305-349.

[80] H. G. Rice, « Classes of recursively enumerable sets and their decision problems », *Transactions of the American Mathematical society*, t. 74, *2*, p. 358-366, 1953.

[81] P. Cousot et R. Cousot, « Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints », in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison et R. Sethi, éd., ACM, 1977, p. 238-252. DOI : 10.1145/512950.512973.

[82] P. Godefroid, N. Klarlund et K. Sen, « DART : directed automated random testing », in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar et M. W. Hall, éd., ACM, 2005, p. 213-223. DOI : 10.1145/1065010.1065036.

[83] S. Pinchinat, F. Schwarzentruber et P. Le Barbenchon, *Logique : fondements et applications : Cours et exercices corrigés*. Dunod, 2022.

[84] J. C. King, « Symbolic Execution and Program Testing », *Commun. ACM*, t. 19, *7*, p. 385-394, 1976. DOI : 10.1145/360248.360252.

[85] M. Davis, G. Logemann et D. W. Loveland, « A machine program for theorem-proving », *Commun. ACM*, t. 5, *7*, p. 394-397, 1962. DOI : 10.1145/368273.368557.

[86] C. W. Barrett et C. Tinelli, « Satisfiability Modulo Theories », in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith et R. Bloem, éd., Springer, 2018, p. 305-343. DOI : 10.1007/978-3-319-10575-8_11.

[87] Y. Shoshitaishvili, R. Wang, C. Salls et al., « SOK : (State of) The Art of War : Offensive Techniques in Binary Analysis », in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, IEEE Computer Society, 2016, p. 138-157. doi : 10.1109/SP.2016.17.

[88] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson et J. Karlsson, « Fault Injection into VHDL Models : The MEFISTO Tool », in *Digest of Papers : FTCS/24, The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing, Austin, Texas, USA, June 15-17, 1994*, IEEE Computer Society, 1994, p. 66-75. doi : 10.1109/FTCS.1994.315656.

[89] J.-B. Machemie, C. Mazin, J.-L. Lanet et J. Cartigny, « SmartCM a smart card fault injection simulator », in *2011 IEEE International Workshop on Information Forensics and Security, WIFS 2011, Iguacu Falls, Brazil, November 29 - December 2, 2011*, IEEE Computer Society, 2011, p. 1-6. doi : 10.1109/WIFS.2011.6123124.

[90] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann et O. Spinczyk, « FAIL* : An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance », in *11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015*, IEEE Computer Society, 2015, p. 245-255. doi : 10.1109/EDCC.2015.28.

[91] M. Hoffmann, F. Schellenberg et C. Paar, « ARMORY : Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries », *IEEE Trans. Inf. Forensics Secur.*, t. 16, p. 1058-1073, 2021. doi : 10.1109/TIFS.2020.3027143.

[92] M. Puys, L. Rivière, J. Bringer et T.-H. Le, « High-Level Simulation for Multiple Fault Injection Evaluation », in *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance - 9th International Workshop, DPM 2014, 7th International Workshop, SETOP 2014, and 3rd International Workshop, QASA 2014, Wroclaw, Poland, September 10-11, 2014. Revised Selected Papers*, J. García-Alfaro, J. Herrera-Joancomartí, E. Lupu et al., éd., sér. Lecture Notes in Computer Science, t. 8872, Springer, 2014, p. 293-308. doi : 10.1007/978-3-319-17016-9_19.

[93] D. Larsson et R. Hähnle, « Symbolic Fault Injection », in *Proceedings of 4th International Verification Workshop in connection with CADE-21, Bremen, Germany, July 15-16, 2007*, B. Beckert, éd., sér. CEUR Workshop Proceedings,

t. 259, CEUR-WS.org, 2007. adresse : https://ceur-ws.org/Vol-259/paper09.pdf.

[94] S. Ducousso, « Aller de la sûreté à la sécurité en analyse de code : le modèle d'attaquant », Thèse de doctorat, Université Grenoble Alpes, 2023.

[95] T. Given-Wilson, A. Heuser, N. Jafri et A. Legay, « An automated and scalable formal process for detecting fault injection vulnerabilities in binaries », *Concurr. Comput. Pract. Exp.*, t. 31, *23*, 2019. doi : 10.1002/CPE.4794.

[96] S. Tollec, V. Hadzic, P. Nasahl et al., « Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults », *IACR Cryptol. ePrint Arch.*, p. 247, 2024. adresse : https://eprint.iacr.org/2024/247.

[97] K. Pattabiraman, N. Nakka, Z. Kalbarczyk et R. K. Iyer, « SymPLFIED : Symbolic program-level fault injection and error detection framework », in *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, IEEE Computer Society, 2008, p. 472-481. doi : 10.1109/DSN.2008.4630118.

[98] L. Goubet, K. Heydemann, E. Encrenaz et R. D. Keulenaer, « Efficient Design and Evaluation of Countermeasures against Fault Attacks Using Formal Verification », in *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, N. Homma et M. Medwed, éd., sér. Lecture Notes in Computer Science, t. 9514, Springer, 2015, p. 177-192. doi : 10.1007/978-3-319-31271-2_11.

[99] P. Cousot, R. Cousot, J. Feret et al., « The ASTREÉ Analyzer », in *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, S. Sagiv, éd., sér. Lecture Notes in Computer Science, t. 3444, Springer, 2005, p. 21-30. doi : 10.1007/978-3-540-31987-0_3.

[100] G. Lacombe, D. Féliot, E. Boespflug et M.-L. Potet, « Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities », *CoRR*, t. abs/2303.03999, 2023. doi : 10.48550/ARXIV.2303.03999.

[101] T. Given-Wilson, N. Jafri et A. Legay, « Combined software and hardware fault injection vulnerability detection », *Innov. Syst. Softw. Eng.*, t. 16, *2*, p. 101-120, 2020. doi : 10.1007/S11334-020-00364-5.

[102] N. L. Binkert, B. M. Beckmann, G. Black et al., « The gem5 simulator », *SIGARCH Comput. Archit. News*, t. 39, *2*, 2011.

[103] J. Proy, K. Heydemann, A. Berzati, F. Majéric et A. Cohen, « A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures : A Secure Software Perspective », in *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, ACM, 2019, 7 :1-7 :10. doi : 10.1145/3339252.3339253.

[104] K. D. Cooper et L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, 2004, isbn : 1-55860-699-8.

[105] J. Y. Yen, « Finding the k shortest loopless paths in a network », *Management Science*, t. 17, *11*, 1971.

[106] A. Hagberg, P. J. Swart et D. A. Schult, « Exploring network structure, dynamics, and function using NetworkX », Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), rapp. tech., 2008.

[107] ARM Limited, *Cortex-M3 Technical Reference Manual*, 2010. adresse : https://documentation-service.arm.com/static/5e8e107f88295d1e18d34714.

[108] C. Fanjas, C. Gaine, D. Aboulkassimi, S. Pontié et O. Potin, « Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing », in *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*, I. Buhan et T. Schneider, éd., sér. Lecture Notes in Computer Science, t. 13820, Springer, 2022, p. 25-44. doi : 10.1007/978-3-031-25319-5_2.

[109] N. A. Quynh et D. H. Vu, « Unicorn : Next generation cpu emulator framework », *BlackHat USA*, t. 476, 2015.

[110] E. Boespflug, A. Bouguern, L. Mounier et M.-L. Potet, « A tool assisted methodology to harden programs against multi-faults injections », *CoRR*, t. abs/2303.01885, 2023. doi : 10.48550/ARXIV.2303.01885.

[111] N. Moro, K. Heydemann, E. Encrenaz et B. Robisson, « Formal verification of a software countermeasure against instruction skip attacks », *J. Cryptogr. Eng.*, t. 4, *3*, p. 145-156, 2014. doi : 10.1007/S13389-014-0077-7.

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Étude de vulnérabilité d'un programme au format binaire en présence de fautes précises et nombreuses

Mot clés : Attaque par injection de fautes, Multi-faute, Analyse statique

Résumé : Les attaques multi-fautes permettent de compromettre la sécurité d'applications prouvées théoriquement robustes, et cela, malgré l'intégration de mécanismes de sécurité. L'évaluation de sécurité pour ce type d'attaque comporte une analyse du programme pour déterminer des vulnérabilités puis une campagne d'injection de fautes sur du matériel. Cependant, considérer plusieurs fautes lors de l'analyse reste un problème ouvert en raison de la taille de l'espace des états fautés à explorer. Ce document vise à étudier les techniques d'évaluation de la sécurité contre des attaques multi-fautes. D'abord, nous explorons faisabilité d'une méthode de détermination de vulnérabilités basée exclusivement sur l'analyse statique. Ensuite, nous étudions une méthode d'identification des paramètres d'injection de fautes afin de faciliter la réalisation de campagne. Des expérimentations ont été menées sur des programmes d'évaluation de code PIN comportant diverses contre-mesures logicielles. Les résultats démontrent l'efficacité de notre approche, avec des attaques comportant jusqu'à huit fautes impactant plus de 80 instructions.

Title: Vulnerability assessment of a binary program in the presence of numerous and precise faults

Keywords: Fault Injection Attack, Multi-Fault, Static Analysis

Abstract: Multi-fault attacks can compromise the security of applications proven as theoretically robust, despite the integration of security mechanisms. The security evaluation for this type of attack involves a program analysis to determine vulnerabilities, followed by a hardware fault injection campaign. However, considering multiple faults during the analysis remains an open problem due to the size of the fault space to explore. This document studies techniques for evaluating the security against multifault attacks. First, we explore the feasibility of a vulnerability determination method based exclusively on static analysis. Secondly, we investigate a method for identifying fault injection parameters to facilitate campaign realisation. Experiments have been carried out on PIN evaluation programs involving various software countermeasures. The results demonstrate the effectiveness of our approach, with attacks involving up to eight faults impacting more than 80 instructions.