Optimizing identification and exploitation of fault injection vulnerabilities on microcontrollers.

Vincent Werner

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Vincent WERNER

Thèse dirigée par **Marie laure POTET**
et co-encadrée par **Laurent MAINGAULT**, CEA

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Optimiser l'identification et l'exploitation de vulnérabilités à l'injection de faute sur microcontrôleurs

Optimizing identification and exploitation of fault injection vulnerabilities on microcontrollers.

Thèse soutenue publiquement le **24 janvier 2022**,
devant le jury composé de :

Madame MARIE-LAURE POTET
Professeur des Universités, GRENOBLE INP, Directrice de thèse
Madame KARINE HEYDEMANN
Maître de conférences HDR, SORBONNE UNIVERSITE, Rapporteure
Monsieur JEAN-MAX DUTERTRE
Professeur, ECOLE NAT SUP MINES ST ETIENNE, Rapporteur
Monsieur GIORGIO DI NATALE
Directeur de recherche, CNRS DELEGATION ALPES, Examinateur
Monsieur ERVEN ROHOU
Directeur de recherche, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE, Examinateur
Monsieur GUILLAUME BOUFFARD
Ingénieur docteur, AGENCE NATIONALE DE LA SECURITE DES SI., Examinateur
Monsieur ROLAND GROZ
Professeur, GRENOBLE INP, Président

Remerciements

Je souhaiterais tout d'abord remercier Marie-Laure Potet pour sa patience et son investissement tout au long de cette thèse. Ses conseils m'ont beaucoup apporté et je la remercie pour la confiance qu'elle m'a accordée pendant ces trois années. Un grand merci aussi à Laurent Maingault pour son encadrement sans faille, pour son expertise technique en injection de faute, mais aussi pour sa bienveillance et ses conseils qui ont su dissiper mes doutes.

J'aimerais ensuite remercier Jean-Max Dutertre, d'une part pour sa participation à mon comité de suivi de thèse, mais aussi pour avoir accepté d'être rapporteur pour cette thèse. Merci également à Karine Heydemann d'avoir accepté de rapporter mon travail. Mes remerciements s'adressent également à Guillaume Bouffard, Giorgio Di Natale, Erven Rohou pour avoir accepté mon invitation à participer au jury de soutenance de ma thèse et enfin à Roland Groz pour m'avoir fait l'honneur de présider ce jury.

Je tiens à exprimer toute ma gratitude auprès de l'équipe du CESTI pour m'avoir accueilli chaleureusement, et ceci dès mon premier jour. Un grand merci à Théo, Aurélien, Yann, Stéphanie, Cécile, Jessy, Nicolas, Véronique, Marielle, Alexis, Frédéric, Antton, Jean-Christophe, David, Damien, Philippe & Philippe et bien sûr Olivier, Olivier, Olivier & Olivier !

Je souhaite aussi remercier tous ceux qui, par leur soutien, leur spontanéité et leur jovialité ont contribué à la réussite de cette thèse. Un grand merci à Marie, dotée d'un talent rare à organiser des soirées chez autruis, épaulée par Vincent, tout aussi doué dans ce domaine. Merci donc à Claire & Benoît pour les barbecues et autres festivités dans le Vercors. Merci à Eleonora pour ses conseils pratiques pour jeune doctorant en panique. Merci à Anne, la meilleure des resp. labo. Encore merci à Laurent, qui en plus d'être un encadrant au top, carbure en vélo. Merci aux supers stagiaires PA, Esther & Karim. Merci à Emrick, l'artiste sous-coté de cette fine équipe. Merci à mes co-thésards de Verimag, Etienne (dernière ligne droite) & Maxime. Également, comment ne pas rendre hommage à mes co-thésards, Loic, Gabriel & Antoine, qui ont partagé mon sort (Julia, Jésus, les cités d'or, etc.). dans notre bureau intimiste, faisant office de salle d'attente du labo.

Pour terminer, un grand merci à mes parents et à ma petite sœur qui m'ont toujours soutenu et encouragé, et enfin merci à toi, Marion, pour toutes tes petites attentions, ton amour et ta patience, et bien sûr pour avoir accepté de m'accompagner à Grenoble, sans quoi cette thèse n'aurait jamais vu le jour.

Résumé

L'injection de faute – notamment par lumière focalisée, impulsion électromagnétique, ou perturbation de la tension d'alimentation – est un moyen extrêmement puissant pour compromettre une application embarquée dans un microcontrôleur. En perturbant physiquement l'environnement de ce dernier, il est possible de modifier son comportement pour extraire des informations secrètes, ou pour contourner des mécanismes de sécurité. Si les attaques par injection de faute peuvent théoriquement nuire à la sécurité des systèmes embarqués, la mise en pratique est souvent difficile car la réussite de ces attaques dépend souvent de la capacité à obtenir un effet de faute particulier, or les effets des fautes sur un microcontrôleur ne sont pas connus à l'avance. Dans le but d'évaluer les vulnérabilités à l'injection de faute, les effets des fautes peuvent être simulés par un outil adéquat. Cependant les effets simulés sont souvent hypothétiques et de nombreuses vulnérabilités identifiées par l'outil seront irréalistes en pratique car l'effet supposé n'existe pas.

Ainsi, l'objectif de cette thèse est de proposer de nouvelles méthodes, techniques et outils pour réduire l'écart entre simulation et expérimentation, dans le but d'optimiser l'identification et l'exploitation de ces vulnérabilités. Pour cela, nous proposons une méthodologie de bout-en-bout, combinant des résultats obtenus expérimentalement et par simulation, dans le but de se concentrer uniquement sur les modèles de faute les plus probables pour améliorer le réalisme des fautes simulées. Cette première approche nous conduira à nous intéresser aux programmes dédiés à la propagation de fautes. Ainsi, nous proposons également plusieurs recommandations pour améliorer la conception de ces derniers, dans le but d'améliorer notre compréhension des fautes sur les microcontrôleurs. Enfin, nous proposons d'utiliser, pour la première fois dans le contexte de l'injection de faute, plusieurs techniques d'optimisation récentes pour optimiser l'identification des meilleurs paramètres d'équipement, afin de faciliter l'exploitation des vulnérabilités à l'injection de faute.

Mot-clés Microcontrôleur - Simulation - Modèle de faute - Optimisation - Injection de fautes

Abstract

Fault injection - for example by focused light, electromagnetic pulse, or power glitch - is an extremely powerful technique for compromising an embedded system. By physically disturbing the environment of the microcontroller, it is possible to modify its behavior to extract secret information, or to bypass security mechanisms. Although fault injection vulnerabilities are a potential threat to the security of many embedded systems, in practice, it is often difficult to identify and exploit them, because fault effects, which depend a lot on the target microcontroller and the fault injection equipment used, are not known in advance. In order to analyze fault injection vulnerabilities, fault effects can be simulated using fault models. However, depending on the model fidelity, some of the vulnerabilities identified may not be exploitable in practice because the modelized effect is not realistic.

Accordingly, the main objective of this thesis is to propose new methods, techniques and tools to reduce the gap between simulation and experimentation, in order to optimize the identification and exploitation of these vulnerabilities. For this, we propose an end-to-end methodology, combining results obtained experimentally and by simulation, in order to infer the most probable fault models to improve the realism of the simulated faults. This first approach will lead us to take an interest in test programs designed to maximize the fault propagation. Thus, we propose original metrics to evaluate these tests, in order to design more efficient ones, and improve our understanding of fault effects. Finally, we propose to use, for the first time in fault injection, two recent optimization techniques to optimize the identification of the best equipment parameters, in order to speed up the exploitation of fault injection vulnerabilities.

Keywords Fault injection - Microcontroller - Optimization - Fault model - Simulation

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 Contexte et motivations

Les microcontrôleurs sont de plus en plus présents dans notre quotidien. Rien qu'une berline contient plus de 50 microcontrôleurs en moyenne [Fle11]. Une tendance qui n'est sans doute pas près de s'inverser, en particulier avec le développement de l'Internet des Objets, des voitures autonomes, des réseaux électriques intelligents, ou encore de la domotique. Cependant, du fait de leur omniprésence, ces composants deviennent une cible de choix pour un attaquant. En effet, une vulnérabilité identifiée sur un microcontrôleur peut potentiellement conduire à la compromission de nombreux systèmes embarqués basés sur ce dernier. D'autant plus que, pour le plus grand bonheur des attaquants, il est généralement très difficile et couteux de corriger les vulnérabilités identifiées, une fois les microcontrôleurs en production.

Par conséquent, pour garantir le bon fonctionnement des systèmes embarqués en milieu hostile, la sécurité des microcontrôleurs passe souvent par l'usage de primitives cryptographiques ou par une limitation (protection) des accès en mémoire. Malheureusement, même en l'absence de bug, cela ne suffit pas à lever toutes les menaces qui pèsent sur les microcontrôleurs. En particulier, l'injection de faute est une technique consistant à perturber délibérément l'environnement physique du microcontrôleur dans le but d'induire une faute pour modifier le comportement de ce dernier. Ainsi, avec l'injection de faute, il est alors possible d'attaquer l'implémentation physique des mécanismes de sécurité, sans reposer sur la présence d'une vulnérabilité logicielle. Pour le développeur, cela peut s'avérer être un véritable casse-tête, notamment du fait qu'il est difficile d'identifier en amont les vulnérabilités à l'injection de faute.

Justement, pour détecter ces dernières, on peut recourir à la simulation ou, à l'inverse, à l'expérimentation. La première approche vise à modéliser les effets des fautes supposés sur le microcontrôleur dans le but d'identifier rapidement et à moindre coût les sections vulnérables de l'application embarquée. La deuxième solution consiste placer le microcontrôleur et l'application embarquée en conditions réelles, en injectant un grand nombre de fautes, de manière à éprouver les mécanismes de sécurité comme le ferait un attaquant. Chaque technique possède ses forces et ses faiblesses. Si l'approche par simulation permet de mener une analyse exhaustive (vis-à-vis des modèles supposés) et d'être intégrable au processus de développement de l'application embarquée pour identifier les vulnérabilités en amont, il se pose la question de la représentativité des résultats de l'analyse pour savoir si les vulnérabilités identifiées seront exploitables. À l'inverse, s'il n'est généralement pas possible de conduire une analyse exhaustive expérimentalement, l'injection de fautes en conditions réelles permet de stresser le comportement du microcontrôleur similairement aux situations rencontrées en milieu hostile. Cependant, bien que l'intérêt de combiner ces deux approches paraisse évident, joindre la simulation et l'expérimentation n'est pas trivial.

1.2 Problématiques

La raison principale limitant le passage de la simulation vers l'expérimentation et vice-versa est la différence de représentation entre les résultats de la simulation selon la modélisation de faute retenue, et les paramètres de l'équipement d'injection faute en fonction du microcontrôleur ciblé. Alors que les résultats de la simulation sont exprimés en fonction de grandeurs liées au niveau d'analyse (registre fauté, instruction ciblée, etc.), les paramètres d'équipement sont exprimés en fonction de grandeurs physiques (position visée sur la puce, puissance du laser, etc.), et il n'y a pas de lien immédiat permettant de relier ces deux représentations différentes. Par conséquent, il est difficile 1) d'exploiter les vulnérabilités identifiées par simulation et 2) de comprendre les vulnérabilités exploitées pendant l'expérimentation.

Une deuxième problématique concerne l'injection de fautes multiples indépendantes, c'est-à-dire l'injection de plusieurs fautes pendant l'exécution de l'application embarquée, dans le but de contourner un mécanisme de sécurité et une éventuelle contre-mesure à l'injection de faute. Si ces attaques sont puissantes, l'explosion combinatoire du nombre de chemins d'attaque avec le nombre de fautes indépendantes injectées complique l'identification et l'exploitation de ces vulnérabilités.

Une autre problématique en injection de faute est de comprendre les effets des fautes sur le microcontrôleur ciblé. Une manière de mettre en évidence ces effets et de charger dans le microcontrôleur ciblé un programme conçu spécifiquement pour maximiser la propagation des effets, dans le but de faciliter l'observation de ces derniers. Dans la littérature, de nombreux programmes de test pour caractériser les fautes ont été proposés afin d'identifier de nouveaux modèles de faute. Cependant, il n'existe pas de formalisation de ces derniers ni de métriques pour les comparer, et ainsi il est difficile d'en choisir un en fonction du besoin.

Enfin, identifier les paramètres d'équipement permettant d'injecter des fautes sur le microcontrôleur ciblé est sans doute l'étape la plus importante. Ce processus, pouvant requérir une grande expertise selon l'équipement utilisé, est long et chronophage. Si plusieurs tech-

niques d'optimisation ont déjà été proposées dans la littérature pour identifier efficacement les meilleurs paramètres, les techniques les plus récentes, déjà utilisées en apprentissage automatique par exemple, n'ont pas été évaluées dans le contexte de l'injection de faute.

1.3 Contributions

Dans cette thèse, nous proposons plusieurs méthodologies et solutions techniques dans le but de réduire l'écart entre simulation et expérimentation. Plus précisément, les contributions de cette thèse sont les suivantes :

- Pour identifier et l'exploiter plus efficacement les vulnérabilités à l'injection de fautes multiples, nous proposons une méthodologie combinant les résultats obtenus expérimentalement avec ceux obtenus par simulation dans le but de se concentrer uniquement sur les modèles de faute les plus probables, pour accélérer la détection de vulnérabilités, améliorer le réalisme des fautes simulées et faciliter la calibration de l'équipement.

- Pour améliorer les tests de caractérisation de fautes, nous formalisons les propriétés importantes de ces tests comme la propagation des effets des fautes, puis nous proposons des métriques de performance pour comparer plusieurs tests populaires dans la littérature. À partir de cette première étude, nous proposons des recommandations pour concevoir ces tests. Avec ces dernières, nous générons plusieurs tests optimaux pour différents modèles de fautes.

- Pour identifier plus rapidement les meilleurs paramètres d'équipement selon le microcontrôleur ciblé, nous proposons de nouvelles techniques d'optimisation basées sur les algorithmes du problème du bandit manchot et l'optimisation bayésienne. Ces techniques récentes sont déjà utilisées pour optimiser des solveurs de problèmes combinatoires très difficiles (e.g. IBM CPLEX [HHLB11]). En plus de ces techniques, nous proposons une méthodologie en deux étapes pour optimiser les paramètres d'équipement indépendamment de l'application embarquée et ainsi accélérer l'identification des meilleurs paramètres.

- Pour simuler plus rapidement les injections de faute, y compris les injections de fautes multiples, de nouvelles fonctionnalités, comme le *multithreading*, ont été rajoutées à l'outil de simulation d'injection de faute `CELTIC`. Pour injecter plus facilement et à bas coût des fautes sur une grande variété de microcontrôleurs, nous proposons l'outil `GLITCH STATION` qui permet d'injecter des fautes par perturbation de la tension d'alimentation. Cet outil permet de contrôler finement la forme de la perturbation via un convertisseur numérique-analogique *low cost*.

1.4 Plan du manuscrit

La suite de ce manuscrit est organisée en six chapitres. Tout d'abord, le chapitre 2 présentera l'état de l'art des attaques par fautes, et en particulier l'histoire de l'injection de

faute, des origines jusqu'à nos jours. Bien que les attaques par fautes visent le plus souvent les algorithmes de chiffrements dans la littérature académique, nous présenterons quelques attaques récentes sur des cibles moins conventionnelles comme une console de jeux vidéos ou un portefeuille électronique de crypto-monnaie. Ensuite, nous détaillerons les principales techniques d'injection de fautes par perturbation. Notamment, nous reviendrons sur la problématique de l'optimisation de l'équipement d'injection de faute et les approches actuelles pour identifier les paramètres induisant le plus de fautes sur la cible. Aussi, dans le but de mieux comprendre les effets des fautes, nous aborderons le processus de caractérisation de fautes et les différentes techniques permettant de recueillir le maximum d'information sur les effets des fautes sur la cible. Nous terminerons ce chapitre avec la modélisation des fautes. En particulier, nous détaillerons les différents niveaux d'analyse possibles, les méthodes proposées pour combler l'écart entre ces niveaux, et le cas de l'injection de fautes multiples indépendantes. Dans la suite de l'état de l'art, au chapitre 3, nous présenterons les outils développés pendant cette thèse, à savoir un outil de simulation d'injection de faute `CELTIC` et un équipement d'injection de faute, la `GLITCH STATION`. Pour ces deux outils, nous décrirons leurs architectures respectives et leurs spécificités.

Le chapitre 4 détaillera notre méthodologie d'identification et d'exploitation de vulnérabilités à l'injection de fautes multiples. Cette dernière sera ensuite illustrée par le biais de plusieurs expériences avec différents microcontrôleurs et techniques d'injection de faute. Puis, nous évaluerons la pertinence de notre approche selon différents critères, notamment la détection de vulnérabilités. Enfin, plusieurs améliorations seront proposées, et certaines d'entre elles seront étudiées de manière approfondie dans les chapitres suivants. En particulier, dans le but d'améliorer la caractérisation de fautes, le chapitre 5 présentera les résultats d'une étude sur un panel de huit tests de caractérisation issus de la littérature en fonction de propriétés qui seront définies. Ces travaux permettront de mettre en évidence les critères à prendre en compte lors de la conception de tests. Ensuite, à partir de cette première étude, nous déterminerons les tests permettant de maximiser la propagation fautes selon un ensemble de modèles de faute. Finalement, ces tests seront utilisés en pratique avec la `GLITCH STATION` pour identifier les modèles de faute les plus probables.

Toujours dans le but d'améliorer la caractérisation de fautes, de nouvelles techniques d'optimisation seront étudiées au chapitre 6. Après avoir détaillé le fonctionnement de ces techniques, nous comparerons les performances de ces dernières avec les techniques d'optimisation fréquemment utilisées lors d'évaluations de sécurité. Également, nous proposerons une nouvelle approche pour accélérer l'identification de vulnérabilités en boîte noire. Puis, avec cette approche et les techniques d'optimisation proposées, nous contournerons un mécanisme de sécurité d'un microcontrôleur récent. Enfin, le chapitre 7 conclura ce manuscrit en résumant les contributions et présentera les perspectives ouvertes par ces travaux.

Chapitre 2

Identification et exploitation de vulnérabilités à l'injection de fautes

Sommaire

2.1 Introduction

Les attaques par injection de faute sont des attaques actives visant l'implémentation physique de la cible. Elles se différencient des attaques logicielles, qui ne visent pas l'implémentation physique, même si cette frontière est de plus en plus fine, notamment depuis les attaques micro-architecturales comme Plundervolt, CLKSCREW et RowHammer [MOG+20, TSS17, KDK+14]. Également, elles se distinguent des attaques par canaux auxiliaires qui sont dites passives, car basées uniquement sur l'observation et l'interprétation de fuites physiques d'information, notamment sous la forme de rayonnement électromagnétique (EM) de la cible.

Les attaques par injection de faute permettent de contourner des mécanismes de sécurité, de prime abord robustes, en exploitant astucieusement l'implémentation de ces derniers. Ces attaques sont d'ailleurs spécialement complexes à prendre en compte lors de la conception d'un produit sécurisé, du fait qu'il soit particulièrement difficile de prévoir les effets des fautes

sur une nouvelle puce. Ainsi, ce chapitre présente l'état de l'art des attaques par injection de faute, mais aussi les techniques et les méthodes d'optimisation d'équipement afin de mieux appréhender les différentes facettes de ces attaques.

Après avoir présenté les origines de l'injection de faute, les différents moyens d'injection et quelques attaques originales à la section 2.2, nous reviendrons à la section 2.3 sur un point crucial en injection de faute, à savoir l'optimisation de l'équipement. Ensuite, à la section 2.4, nous étudierons les effets des fautes sur différentes architectures et les approches proposées pour mieux les comprendre. Puis, à la section 2.5, nous détaillerons la modélisation et la simulation des fautes ainsi que les méthodologies existantes visant à combler l'écart entre différents niveaux d'analyse. Enfin, la section 2.6 conclut ce chapitre en positionnant les objectifs de cette thèse.

2.2 Les attaques par injection de faute

Dans cette section, nous revenons sur les notions de faute, d'erreur et de défaillance, puis nous nous intéressons à l'injection de faute, de son origine jusqu'à son utilisation en cybersécurité. Ensuite, nous revenons sur les moyens d'injection connus. Enfin, nous terminons par présenter quelques attaques originales.

2.2.1 Faute, Erreur, Défaillance

On désigne sous le terme d'*erreur* une déviation par rapport au comportement nominal. Une erreur est la manifestation d'une *faute*, dans notre cas induite par une *perturbation*. Lorsqu'une erreur se propage jusqu'à être observable, on parle alors de *défaillance*. Si une faute est nécessaire pour causer une erreur, toutes les fautes ne résulteront pas en défaillance. Certaines erreurs peuvent être masquées, c'est-à-dire corrigées avant de pouvoir se propager. La figure 2.1 résume schématiquement la relation entre faute, erreur et défaillance [Muk11, ALR01, Lap92].

Les fautes peuvent être classifiées selon la durée de leurs effets. En sécurité, on retrouve principalement des fautes *transitoires*, c'est-à-dire des fautes temporaires, n'apparaissant qu'à la suite d'une perturbation. On les distingue des fautes *permanentes*, qui lorsqu'elles surviennent, persistent dans le temps. Dans cette thèse, on traitera exclusivement de fautes transitoires.

Les effets des fautes est évidemment un aspect très important dans le contexte de l'évaluation de la sécurité aux attaques par fautes. Il est fréquent d'utiliser des modèles de faute pour décrire les effets des fautes sur le microcontrôleur ciblé. Nous reviendrons plus en détail sur ce sujet à la section 2.5.

2.2.2 L'injection de faute, des origines à nos jours

L'injection de faute est le fait d'introduire une faute volontairement dans un système. Cette approche est utilisée dans le domaine de la sûreté, dès les années 1970, pour évaluer la fiabilité et la résilience d'un circuit [WP83, AAA+90, CP95]. Le but est de reproduire plus

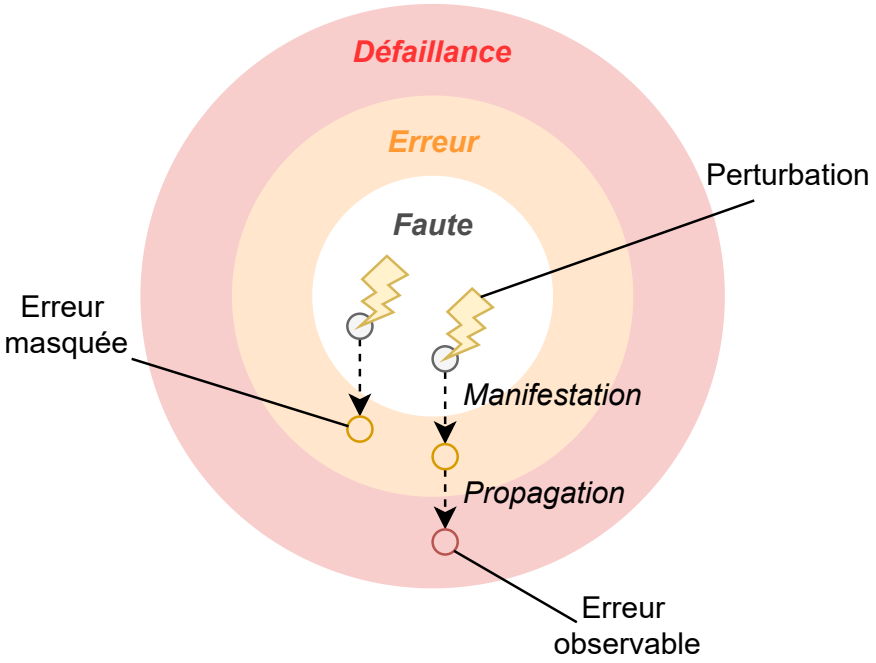


FIGURE 2.1 – Vue schématique de la relation entre faute, erreur et défaillance

facilement des fautes dues à un vieillissement prématuré, ou aux perturbations environnementales telles que les fluctuations de tension ou les interférences électromagnétiques.

Ce n'est que bien plus tard, dans les années 1990, que l'injection de faute apparait dans le domaine de la sécurité, avec les premières attaques sur les cartes à puce utilisées par les systèmes de télévision à péage [AK96]. L'injection de faute permet ici de perturber délibérément le comportement du système afin de contourner les mécanismes de sécurité et de récupérer, dans le cas de la télévision à péage, la clé nécessaire pour déchiffrer le contenu payant. Mais c'est surtout à cette même période que Boneh et al. [BDL97] théorisent la première attaque par faute, aujourd'hui connue sous le nom d'attaque BellCore, permettant de retrouver la clé privée de certaines implémentations de l'algorithme de chiffrement asymétrique RSA. Cette attaque exploite la différence entre deux signatures d'un même message, l'une étant fautée, pour factoriser le modulo n et trouver la clé.

Le terme d'analyse différentielle de fautes (*Differential Fault analysis*, DFA) sera d'ailleurs introduit par Biham et al. [BS97], et regroupe l'ensemble des attaques exploitant des différences entre les résultats corrects et fautés obtenus. À l'inverse, l'analyse de faute par collisions (*Collision Fault analysis*, CFA, [BK06]) exploite le fait que deux chiffrés, dont l'un fauté, soient identiques. Une variante de cette classe d'attaque consiste à injecter des fautes sans effet (*Ineffective Fault analysis*, IFA, [Cla07]).

Pour aller plus loin, l'analyse de faute par outil statistique (*Statistical Fault analysis*, SFA, [FJLT13]) permet de relâcher les contraintes sur la connaissance précise du modèle de faute, en comparaison avec l'analyse différentielle de fautes. Plus récemment, en généralisant les résultats des travaux sur l'IFA, et en utilisant les outils statistiques de la SFA, il est possible de retrouver la clé privée avec des fautes sans effet, en contournant les contremesures

ne prenant pas en compte ce type de fautes (*Statistical Ineffective Fault analysis*, SIFA, [DEK+18]).

En réponse à ces nouvelles classes d'attaque, des contre-mesures furent proposées (e.g. [CJ05, Gir05]). Si ces contre-mesures permettaient de se protéger contre une seule injection de faute par exécution, elles ne prenaient pas en considération le cas de plusieurs injection de faute par exécution. En injectant une première faute pour corrompre le résultat puis une deuxième pour contourner la contre-mesure, Kim et al. [KQ07] parviennent, avec une attaque BellCore modifiée, à retrouver la clé privée, réussissant, par la même occasion, la première attaque multi-faute expérimentalement. Depuis, plusieurs travaux démontrent de l'importance de prendre en compte l'injection de faute multiple [TK10, BDSG+14, BH15, SHS16, VdHOGT21]. Cependant, ces attaques restent difficiles à identifier, en particulier pour les développeurs de systèmes sécurisés. Nous reviendrons sur cette problématique à la section 2.5.

Si les attaques par fautes sont souvent associées aux systèmes cryptographiques dans la littérature (AES, RSA, etc.), nous présenterons des exemples d'attaques sur des systèmes non-cryptographiques à la sous-section 2.2.4. Enfin pour conclure cet exposé sur les attaques par injection de faute, le dernier point important à aborder concerne les techniques d'injection de faute.

2.2.3 Techniques d'injection de faute par perturbation

Il existe une grande variété de techniques pour injecter des fautes sur un composant. Pendant cette thèse, on s'est intéressé plus particulièrement aux techniques d'injection de faute par perturbation. La particularité de ces dernières est d'altérer l'environnement du composant localement pour modifier son comportement. On parle aussi d'attaque semi-invasive car il peut être nécessaire de préparer la cible, en la décapsulant par exemple, c'est-à-dire retirer le boitier pour avoir un accès direct au silicium. Si les attaques semi-invasives peuvent entrainer par inadvertance ou malchance une destruction de la puce, les attaques invasives conduisent nécessairement à une détérioration du circuit. Dans la suite de cette section, nous détaillons uniquement les techniques d'injection de faute par perturbation.

2.2.3.1 Perturbation de la tension d'alimentation

Cette technique est l'une des premières utilisées pour générer des fautes dans le but de compromettre un système [AK96, AK97]. Le principe consiste à faire varier la tension d'alimentation de la cible (figure 2.2), aussi appelé *Power Glitch*. En s'éloignant brutalement de la tension nominale du composant pendant un court instant, il est possible de modifier l'exécution de l'application ciblée. L'équipement nécessaire pour contrôler la forme de l'impulsion électrique générée est généralement peu onéreux. Si cette technique requiert quelques connaissances, elle reste néanmoins accessible, car ne nécessitant pas d'outillage spécialisé. La faute produite est transitoire, globale et difficilement contrôlable. Cependant, nous verrons qu'il est possible d'améliorer le contrôle et la précision des impulsions électriques générées en utilisant un outillage avancé au chapitre 3.

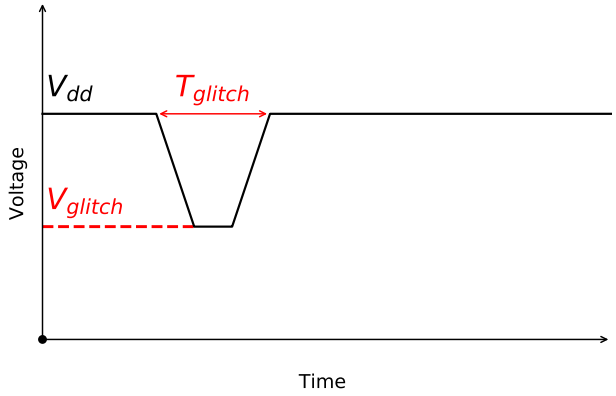


FIGURE 2.2 – Perturbation de la tension d'alimentation. La tension nominale V_{dd} est abaissée à une tension V_{glitch} pendant une durée T_{glitch}.

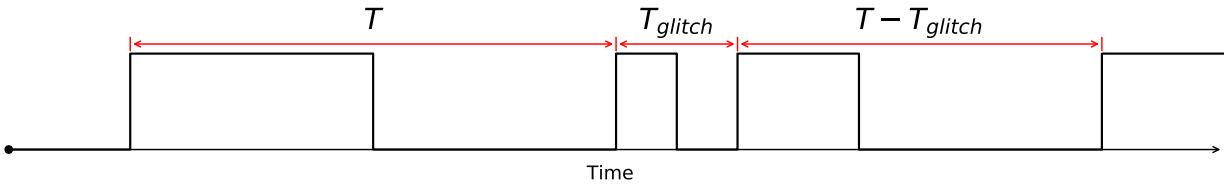


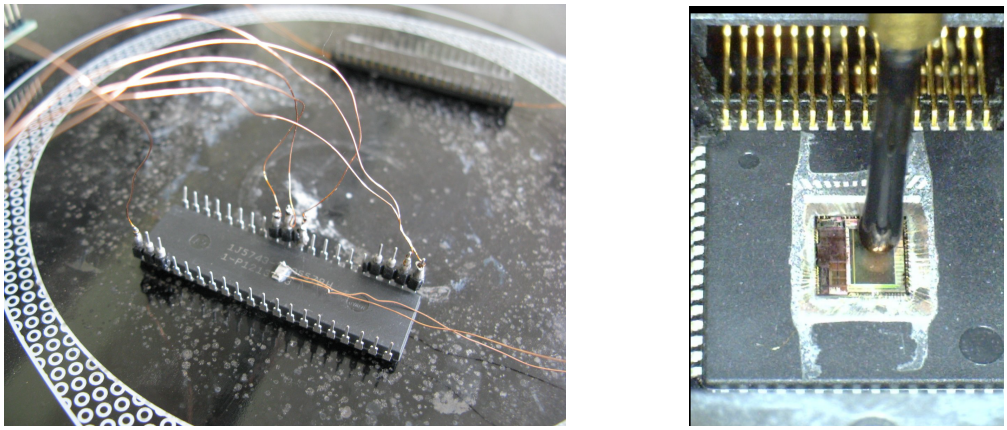
FIGURE 2.3 – Perturbation du signal d'horloge. La période nominale T du signal d'horloge est raccourcie d'une période T_{glitch}.

2.2.3.2 Perturbation du signal d'horloge

Similairement, perturber le signal d'horloge du composant est une autre technique peu coûteuse pour induire une faute pendant l'exécution d'une application (figure 2.3). L'approche classique consiste à modifier subitement la fréquence d'horloge [BGV11], aussi appelée *Clock Glitch* de manière à induire des défauts de propagation de signaux. Cette technique d'injection ne nécessite pas d'outillage onéreux ou spécialisé, typiquement, un FPGA permet à la fois de générer et de perturber le signal d'horloge de la cible. Cependant, cette technique requiert une cible dont l'horloge est contrôlée par un signal externe, ce qui n'est pas toujours le cas. De plus, comme pour les perturbations de la tension d'alimentation, la faute produite est transitoire, globale et difficilement contrôlable.

2.2.3.3 Température

Augmenter la température au delà de la plage de fonctionnement de la cible permet d'induire des fautes. Les fautes générées peuvent être locales et permanentes [Sko09] ou globales et transitoires [HS13]. L'expertise technique et le coût d'équipement varie avec l'approche utilisée. L'approche classique (figure 2.4a) ne requiert pas d'outillage spécialisée (une plaque chauffante suffit). Cependant, la précision et le contrôle est très limité car il est difficile de provoquer des variations de température localisées, rapides et contrôlées avec une plaque chauffante. Contrairement aux perturbations précédentes, le risque de détériorer le



(a) (b)

FIGURE 2.4 – (a) Plaque chauffante avec deux capteurs PT100 mesurant la température arrière et avant d'un ATmega162 [HS13] (b) Sonde d'injection électromagnétique au-dessus d'un microcontrôleur décapsulé [DDRT12].

circuit est élevé.

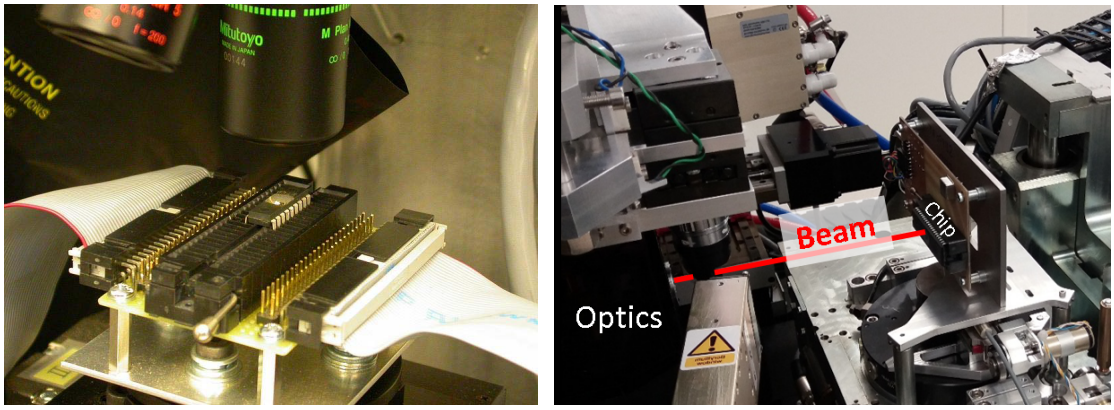
2.2.3.4 Impulsion électromagnétique

Perturber la cible à l'aide d'impulsions électromagnétiques est une technique populaire pour injecter des fautes transitoires et localisées. Depuis la première attaque BellCore avec cette technique, proposée par Schmidt et al. [SH07], l'injection de faute par impulsion électromagnétique (EM) est particulièrement bien couverte dans la littérature [DDRT12, MDH+13, OGST+14, MBB16, TBE+21]. Cette approche consiste à générer un champ magnétique, le plus souvent avec une bobine et un courant impulsionnel. Noter que cette attaque peut se réaliser sans décapsuler la carte, mais les résultats sont meilleurs en ouvrant le composant (figure 2.4b). Le coût d'équipement et l'expertise technique demandée est généralement plus élevée que les approches précédentes. Cela est à nuancer car Abdellatif et al. [AH20] ont récemment proposé un équipement bon marché.

2.2.3.5 Lumière focalisée

Lorsque le prix de l'équipement et l'expertise ne sont plus des contraintes et que seuls comptent les résultats, l'injection de faute par lumière focalisée est une technique permettant d'induire avec précision des fautes transitoires dans un circuit. L'approche consiste à tirer parti de l'énergie apportée par le rayonnement lumineux pour commuter l'état des transistors ciblés (figure 2.5a). La première attaque BellCore par lumière focalisée est réalisée en 2002 par Skorobogatov et al. [SA02] avec un simple laser classe 2[1] à 8$. Depuis l'équipement a évolué, avec notamment l'utilisation de laser en proche infrarouge [DRPR19] ou d'équipement laser

[1]Lasers à rayonnement visible (400 à 700 nm de longueur d'onde), et d'une puissance inférieure ou égale à 1 mW. Protection de l'œil assurée par le réflexe palpébral.



(a) (b)

FIGURE 2.5 – (a) Équipement d'injection de faute par lumière focalisée [Sko10] (b) La ligne de lumière de tomographie ID16B de l'ESRF utilisée pour induire des fautes par Rayon X sur un ATmega1284P [ABC+17].

double spot [SHS16]. La facture est aussi plus salée : il n'est pas rare de dépasser les 100k€ selon l'outillage choisi [BJ15].

2.2.3.6 Rayon X

Si l'injection par lumière focalisée n'est pas suffisamment élitiste pour vous, pourquoi ne pas essayer l'injection par nanofaisceaux à rayons X ? En 2017, Anceau et al. [ABC+17] propose d'utiliser la ligne de lumière de tomographie ID16B de l'ESRF (*European Synchrotron Radiation Facility*) pour injecter des fautes semi-permanentes sur un ATmega1284P (figure 2.5b). Ce type d'équipement est extrêmement rare et onéreux. Un faisceau intense, d'une dizaine de nanomètres seulement, est focalisé sur la mémoire flash, EPROM et RAM du circuit ciblé. Les fautes sont dites semi-permanentes car l'effet produit est réversible, et un simple traitement thermique permet de supprimer l'effet induit. La faute est très localisée, il est possible de viser précisément un seul transistor.

2.2.3.7 Comparatif

Pour conclure sur les techniques d'injection de faute, le tableau 2.1 compare ces dernières en fonction de la précision et du contrôle sur les fautes induites, l'expertise technique requise et le coût de l'outillage. Noter que de récents travaux proposant diverses améliorations pour réduire le coût de l'outillage ou augmenter la précision des fautes injectées ([BFP19, AH20, CPHR21]) bouscule cette classification. Aussi, certaines variantes, qui ne sont pas prises en compte dans ce comparatif, permettent d'induire des fautes plus précises. Par exemple, il est possible de générer des fautes par température localisée avec un laser au lieu d'utiliser une plaque chauffante.

Technique d'Injection	Précision & Contrôle	Expertise Technique	Coût Équipement
Température	Très Faible	Très Faible	Très Faible
Power Glitch	Faible	Faible	Faible
Clock Glitch	Faible	Faible	Faible
Impulsion électromagnétique	Modéré	Modérée	Modéré
Lumière Focalisée	Élevé	Modérée	Élevé
Rayon X	Très Élevé	Très Élevée	Très Élevé

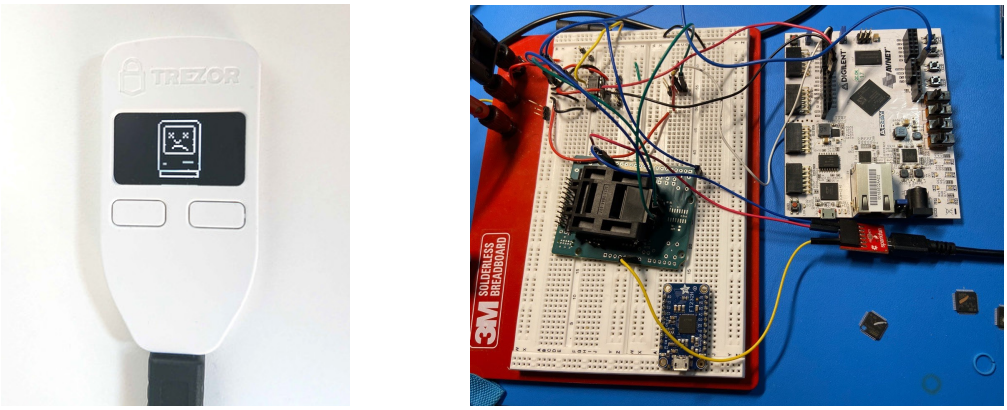
TABLE 2.1 – Comparaison des techniques d'injection de faute par perturbation selon la précision et le contrôle sur les fautes induites, l'expertise technique et le coût de l'équipement.

2.2.4 Sélection d'exemple d'attaques

Précédemment, nous avons évoqué les attaques par faute sur des algorithmes de chiffrement (e.g. attaque BellCore) pour compromettre des accélérateurs cryptographiques. Comme les travaux de cette thèse ne se restreignent pas aux crypto-systèmes, nous présentons quelques attaques supplémentaires sur des consoles de jeux vidéos ou des portefeuilles électroniques de crypto-monnaie. Ces attaques reposent sur la compromission des mécanismes de protection limitant les fonctionnalités du chargeur de démarrage.

2.2.4.1 Chargeur de démarrage et mécanismes de protection

Le chargeur de démarrage ou *bootloader* est une application, le plus souvent préprogrammé directement sur le microcontrôleur en usine, permettant à l'utilisateur de lire et écrire en mémoire. Des mécanismes de protection, désignés sous le nom de *Code Readout Protection* (CRP) [VdHOGT21], peuvent être activés pour restreindre l'accès en lecture et en écriture à partir du *bootloader*. Les CRP proposent généralement plusieurs niveaux de protection, allant de la limitation des fonctionnalités du *bootloader* jusqu'à la désactivation complète et irréversible de ce dernier. En compromettant ces mécanismes, un individu malveillant peut récupérer des informations sensibles (e.g. clé privée) ou étudier la conception d'un produit concurrent (rétro-ingénierie). Par ailleurs, la compromission de CRP fait partie des domaines de recherche très actifs ces dernières années, par exemple, en perturbant les cellules mémoires contenant la valeur du niveau de protection CRP par lumière focalisée ([OT17]), ou bien en contournant les vérifications du niveau de protection CRP avec des *power glitches* [Ces16, Ger17, BFP19, RDN19, VdHOGT21]. Une fois ces vulnérabilités connues, elles sont difficiles (impossibles) à corriger une fois le microcontrôleur commercialisé. Cela peut être critique, en particulier lorsque que l'application finale a pour but la protection de données sensibles.



(a) (b)

FIGURE 2.6 – (a) Trezor One [AGH19] (b) Equipement d'injection de *power glitch* dans le but d'attaquer le Trezor One [Kra20].

2.2.4.2 Portefeuille électronique de crypto-monnaie

Un portefeuille électronique de crypto-monnaie ou *hardware wallet* est un support physique permettant de stocker les clés privées et publiques de l'utilisateur, nécessaire pour envoyer et recevoir des crypto-monnaies. En dérobant la clé privée d'un utilisateur, un individu malveillant peut se saisir de la fortune amassée sur cette dernière. Mais alors, quel rapport avec l'injection de faute ? En 2019, deux équipes [AGH19, Kra20] ont montré que le *hardware wallet Trezor One* était vulnérable à l'injection de faute (figure 2.6). En abaissant le niveau de protection CRP du microcontrôleur embarqué dans le *Trezor One*, le *bootloader* normalement inaccessible, peut être réactivé. Il est alors possible de lire en SRAM et de récupérer la phrase mnémonique de l'utilisateur (*seed*). L'individu malveillant, une fois en possession de la *seed* de l'utilisateur, peut récupérer l'ensemble des fonds associés aux adresses gérées par le portefeuille. Cette attaque est rendue possible car le microcontrôleur contenant les données sensibles de l'utilisateur n'est pas sécurisé. Les microcontrôleurs sécurisés sont conçus pour être robustes aux attaques physiques comme les injections de faute et sont conseillés pour ce type d'application.

2.2.4.3 Consoles de jeux vidéos

L'injection de faute peut être utilisée pour exploiter des vulnérabilités logicielles plus classiques. Un très bon exemple récent est sans doute celui de Yifan Lu [Lu19] qui, en 2019, réussit à extraire l'intégralité de la mémoire système de la *PlayStation Vita* (une console portable de jeux vidéos) contenant, entre autres, le *bootloader* de cette dernière. Lors du démarrage de la console, le secteur 0 de la partition est lu, contenant la zone d'amorce (*Master Boot Record*, MBR). Ce dernier contient un champ contenant la taille du *bootloader* et sa position dans la partition. L'auteur supposa (après quelques expérimentations) que le bootloader était vraisemblablement lu et chargé dans un tableau de taille fixe, après une vérification de la taille du *bootloader* pour éviter un dépassement de tampon (*buffer overflow*). Avec une injection de faute au bon moment, il est possible de sauter cette vérification, et

	Paramètres				
	X	Y	Z	$Pulse_{duration}$	$Pulse_{power}$
Description	Position X de la cible	Position Y de la cible	Position Z de la focale	Durée de la pulsation	Puissance crête de la pulsation
Unité (SI)	Mètre (m)	Mètre (m)	Mètre (m)	Seconde (s)	Watt (W)

TABLE 2.2 – Les paramètres d'un équipement d'injection Laser.

d'exploiter un *buffer overflow* dans le but d'exécuter un code tiers afin d'extraire le contenu de la mémoire système de la *PlayStation Vita*. Une partie importante du travail réalisé pour mener à bien cette attaque consiste à trouver les bons paramètres de l'équipement d'injection de faute. C'est ce que nous allons voir dans la prochaine section.

2.3 Optimisation des paramètres d'attaque

Perturber un microcontrôleur sans que ce dernier s'interrompt inopinément se joue souvent à des détails, quelques mV avec des *power glitches* par exemple. Ainsi, le nerf de la guerre en injection de faute est de trouver les paramètres d'attaque, c'est-à-dire d'une part, les paramètres d'équipement qui induisent des fautes sur le microcontrôleur ciblé, et d'autre part, les délais d'injection qui ciblent les zones critiques de l'application embarquée. Cette section présente les différentes techniques et approches utilisées pour trouver les meilleurs paramètres d'attaque.

2.3.1 Paramètre d'attaque

Dans ce manuscrit, les paramètres d'attaque désignent l'ensemble des paramètres expérimentaux pour reproduire physiquement une attaque, à savoir, les paramètres d'équipement et les délais d'injection.

2.3.1.1 Paramètre d'équipement

Les paramètres d'équipement désignent l'ensemble des réglages spécifiques pour un équipement d'injection de faute donné. La taille et le nombre de dimensions (i.e. paramètres) de l'espace des réglages possibles est variable d'un équipement à un autre. Par exemple, si on retrouve seulement 2 paramètres pour une grande majorité des équipements d'injection de *power glitch* (figure 2.7), un équipement d'injection Laser standard se configure généralement avec 5 paramètres (tableau 2.2). Ainsi, en fonction de l'équipement utilisé, la difficulté à identifier les paramètres induisant des fautes sur la cible n'est pas la même. De manière générale, elle est proportionnelle aux nombres de combinaisons de paramètres possibles pour l'équipement considéré. Par exemple, si un équipement d'injection de *power glitch* se limite à 10^4 combinaisons [PBBJ15], un équipement d'injection EM peut atteindre 10^{12} combinaisons [MSPB19].

	Paramètre	
	V_{glitch}	T_{glitch}
Description	Amplitude du glitch	Durée du glitch
Unité (SI)	Volt (V)	Seconde (s)

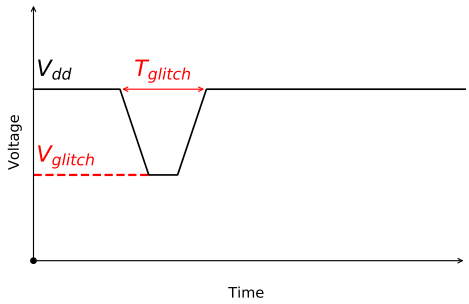


FIGURE 2.7 – Les paramètres d'un équipement d'injection de *power glitch*.

2.3.1.2 Délai d'injection

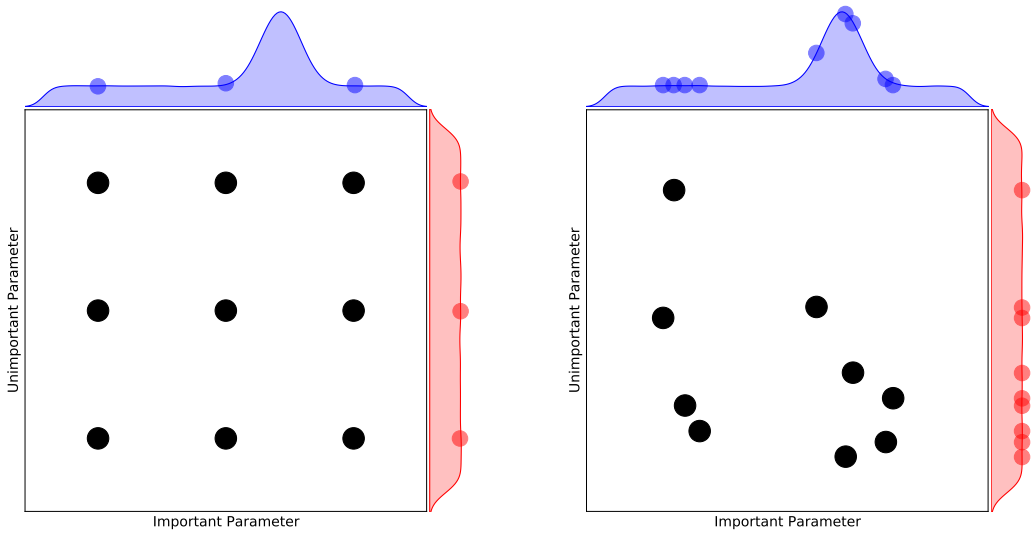
Le délai d'injection est un paramètre commun à tous les équipements d'injection de faute, dès lors que l'on souhaite injecter une faute transitoire. Le délai d'injection désigne le temps écoulé entre un point de référence et l'injection de la faute. Par exemple, le pin *reset* du circuit est souvent utilisé comme point de référence pour compromettre des CRP ([BFP19, RDN19]). Autre exemple, au chapitre 6, à la section 6.6, le point de référence est la fin de transmission UART. Noter qu'en fonction du point de référence choisi, il peut avoir des variations temporelles (*jitter*) entre deux exécutions ce qui peut gêner à la répétabilité des attaques. Autre point important, pour induire des fautes sur le microcontrôleur cible, il est primordial de trouver les bons paramètres d'équipement, mais pas nécessairement les délais d'injection. Pour cette raison, nous verrons dans la suite que certaines approches séparent le délai d'injection du reste des paramètres d'équipement ([CPB+13]), pour réduire l'espace de recherche.

2.3.1.3 Recherche exhaustive

La recherche exhaustive consiste à tester toutes les combinaisons de paramètres d'attaque possibles. Or, l'espace des paramètres d'attaque (i.e. paramètres d'équipement *et* délai) est généralement bien trop large pour être couvert entièrement avec une recherche exhaustive. Ainsi plusieurs techniques ont été proposées pour contourner ce problème. On retrouve d'une part les techniques d'optimisation accélérant l'identification des meilleurs candidats, et d'autre part les techniques de réduction de l'espace, qui visent à délimiter et réduire significativement l'espace des paramètres en identifiant les zones d'intérêt. Ces approches seront respectivement détaillées à la sous-section 2.3.2 et sous-section 2.3.3.

2.3.2 Technique d'optimisation

Les techniques d'optimisation dans le contexte de l'injection de faute reposent généralement sur des approches simples et bien connues : la recherche par quadrillage et la recherche aléatoire. Ce n'est que récemment que des stratégies plus évoluées ont émergé, notamment basées sur des algorithmes métaheuristiques. Cependant, peu importe la stratégie choisie, le but reste le même : trouver le plus rapidement des combinaisons de paramètres d'attaque



(a) (b)

FIGURE 2.8 – Exploration d'un espace de deux paramètres, dont l'un a peu d'effet sur le microcontrôleur, avec a) une recherche par quadrillage versus b) une recherche aléatoire [BB12].

induisant une faute sur le microcontrôleur. Tout d'abord, nous expliquerons la recherche par quadrillage et la recherche aléatoire ainsi que leurs inconvénients. Ensuite, nous reviendrons sur les différents algorithmes métaheuristiques qui ont été proposés dans la littérature dans le contexte de l'injection de faute.

2.3.2.1 Recherche par quadrillage

La recherche par quadrillage, ou *Grid Search* (GS), est une recherche semi-exhaustive sur une plage de valeurs prédéterminée et progressivement affinée. Bien que GS soit suffisante lorsque l'espace de recherche est de faible taille, cette technique est inefficace pour un espace de paramètres de grande dimension, car le nombre de configurations évaluées augmente de façon exponentielle avec le nombre de paramètres considérés (fléau de la dimension [Bel61]). L'autre inconvénient de la recherche par quadrillage est illustrée à la figure 2.8. Dans cet exemple, la recherche par quadrillage échantillonne à intervalle régulier chaque paramètre de l'espace. Or l'un des paramètres a peu d'effet sur le microcontrôleur. En explorant seulement trois valeurs différentes du paramètre important sur 9 essais, la recherche par quadrillage manque la région intéressante de l'espace des paramètres.

2.3.2.2 Recherche aléatoire

La recherche aléatoire, ou *Random Search* (RS), est une autre stratégie simple qui corrige l'inefficacité de la recherche par quadrillage, illustrée précédemment. En effet, comme illustré à la figure 2.8, RS permet d'explorer 9 valeurs différentes du paramètre important et

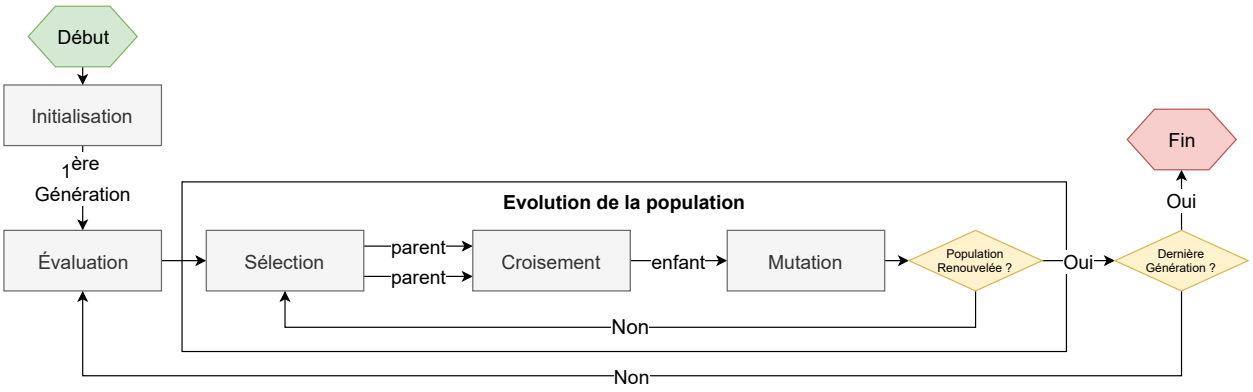


FIGURE 2.9 – Vue schématique des étapes principales d'un algorithme génétique.

par la même occasion, d'explorer la région intéressante. Cependant, même si RS est un peu plus efficace que GS pour explorer des espaces de grande dimension [BB12], RS et GS ont un défaut commun, à savoir qu'elles sélectionnent la prochaine configuration à tester indépendamment des résultats précédents, ce qui a pour conséquence de gâcher de nombreuses évaluations sur des configurations sans intérêt particulier.

2.3.2.3 Algorithme métaheuristique

D'autres approches ont été proposées pour identifier les paramètres d'attaque en fonction de l'équipement utilisé et du microcontrôleur ciblé. En particulier, les algorithmes métaheuristiques permettent d'explorer plus efficacement l'espace des paramètres que GS et RS. Les algorithmes métaheuristiques désignent les algorithmes d'optimisation s'inspirant de théories du vivant, le plus souvent stochastiques et itératifs, [GT13], et visant à résoudre des problèmes difficiles pour lesquels on ne connaît de méthode classique plus efficace.

En injection de faute, on retrouve d'une part l'algorithme génétique, ou *Genetic Algorithm* (GA), un algorithme métaheuristique populaire basé sur la théorie de l'évolution, déjà utilisé pour optimiser les paramètres des équipements dinjection EM [MSPB19], mais aussi des équipements d'injection de *power glitch* [BFP19, PBJC14, CPB+13]. D'autre part, Picek et al. [PBBJ15] proposent un algorithme mémétique pour calibrer un équipement d'injection de *power glitch*, qui est une extension du GA classique avec une technique de recherche locale.

Un algorithme génétique est un algorithme itératif s'articulant autour de quatre étapes, l'évaluation, la sélection, le croisement et la mutation. Le but est de faire évoluer une population de candidats, sélectionnés initialement aléatoirement, pour s'approcher progressivement de la solution optimale du problème, comme illustré à la figure 2.9.

Initialisation Les individus sont sélectionnés aléatoirement dans l'espace des paramètres d'attaque, pour former la 1ère génération.

Évaluation La valeur sélective (*fitness*) de chaque individu de la population est évaluée. La valeur sélective d'un individu est une note correspondant à son adaptation au problème.

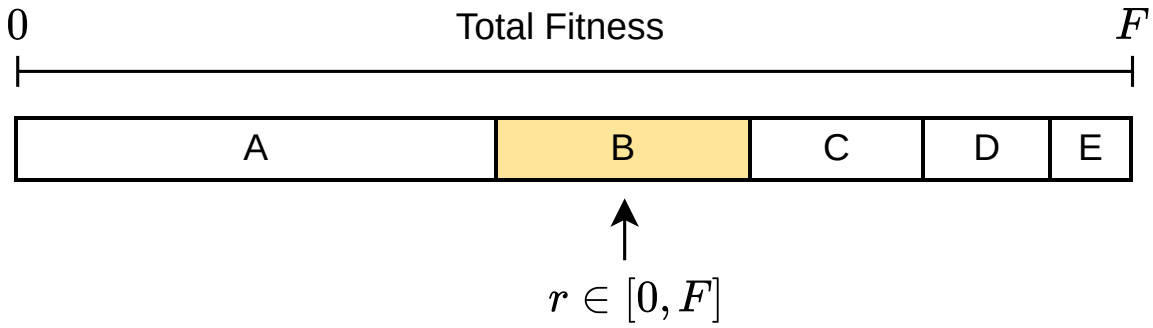


FIGURE 2.10 – Sélection par tirage à la roulette avec r tiré aléatoirement entre 0 et le total F des valeurs sélectives de la population. Dans cet exemple, l'individu B est sélectionné [Hat07].

Par exemple, en calculant la moyenne empirique sur plusieurs essais d'obtenir un résultat fauté avec l'individu évalué $\frac{\#\{\text{résultats fautés}\}}{\#\{\text{essais}\}}$.

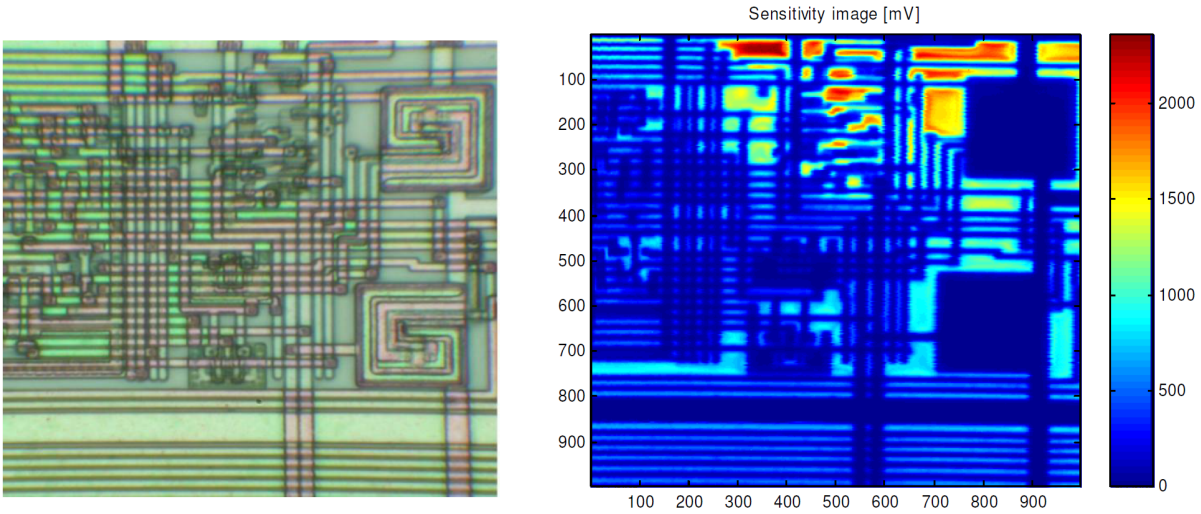
Sélection Deux parents sont sélectionnés dans l'ensemble de la population en suivant une stratégie de sélection. Les techniques de sélection les plus populaires sont la sélection par tirage à la roulette et la sélection par tournoi. La *sélection par tirage à la roulette* sélectionne les individus proportionnellement à leur valeur sélective (figure 2.10). La *sélection par tournoi* consiste à sélectionner des individus au hasard et prendre le meilleur parmi ceux sélectionnés.

Croisement Les deux individus parents, sélectionnés précédemment, sont croisés pour former un individu enfant. Encore une fois, plusieurs stratégies existent. Par exemple, avec un *croisement uniforme*, chaque valeur de paramètre de l'enfant est choisie parmi l'un ou l'autre parents avec une probabilité égale.

Mutation Des mutations sont appliquées aux valeurs des paramètres de l'enfant. Une fois n'est pas coutume, il existe de nombreux opérateurs de mutation. En particulier, l'opérateur de mutation gaussien ajoute une valeur aléatoire suivant une distribution gaussienne pour chaque valeur de paramètre de l'enfant.

Ce processus itératif continue jusqu'à atteindre la dernière génération, fixée par l'utilisateur. Les algorithmes génétiques sont flexibles et s'adaptent facilement à une grande variété de problèmes, ce qui permet de les utiliser pour optimiser les paramètres d'attaque dans plusieurs situations différentes.

Cependant, ces derniers souffrent de deux inconvénients majeurs. D'une part, entre les multiples stratégies de sélection, opérateurs de mutation, et technique de croisement, mais aussi la taille de la population et le nombre de générations, les algorithmes génétiques peuvent être complexes à paramétrer. D'autre part, qui est une conséquence du point précédent, les algorithmes génétiques, lorsque mal configurés, peuvent souffrir d'une convergence prématurée vers une solution sous-optimale.



(a) (b)

FIGURE 2.11 – (a) Image optique (grossissement 500×) et (b) balayage OBIC d'un microcontrôleur PIC16F84A [Sko05].

2.3.2.4 Apprentissage profond

Plus récemment, Wu et al. [WRBBP20], ont proposé une méthode pour caractériser la sensibilité d'un microcontrôleur sécurisé aux perturbations laser, en se basant sur de l'apprentissage profond, pour ajuster la largeur de l'impulsion et la puissance du laser. Cette technique souffre du même problème que les algorithmes génétiques à savoir comment ajuster les hyperparamètres du modèle (nombre de couches cachées, nombre de neurones par couche, etc.). Par ailleurs, les modèles d'apprentissage profond sont généralement si complexes à optimiser, qu'il faut recourir à des techniques d'optimisation (par exemple... un GA [YS20]).

2.3.3 Réduction de l'espace des paramètres

Lorsque applicable, réduire l'espace des paramètres en se concentrant uniquement sur les zones d'intérêt permet d'accélérer significativement l'identification des meilleurs paramètres d'attaque. Cette partie détaille les approches proposées à cette fin.

2.3.3.1 Positions

L'identification des positions sur la puce jouant un rôle critique lors de l'exécution de l'application cible est une priorité en injection EM ou Laser. Plusieurs approches peuvent être utilisées dans ce but. Par exemple, en utilisant un microscope électronique à balayage, Courbon et al. [CLMFT14] détecte les régions de la puce qui seront les plus sensibles aux perturbations laser. Il est aussi possible de mesurer le courant induit par faisceau optique, aussi appelé *Optical beam-induced current* (OBIC), en s'en servant comme technique d'imagerie, ce qui permet ensuite d'identifier les zones actives du microcontrôleur (figure 2.11), et réduire considérablement les zones à cibler avec le laser [Sko05, SFR+15]. Madau et al.

Travaux Existants	Technique d'optimisation	Réduction de l'espace	Technique d'injection
[PBBJ15]	Algorithme Mémétique	✗	Power Glitch
[MSPB19]	Algorithme Génétique	✗	EM
[BFP19, PBJC14]	Algorithme Génétique	✗	Power Glitch
[CPB+13]	Algorithme Génétique	Délai d'injection	Power Glitch
[WRBBP20]	Apprentissage Profond	✗	Laser
[MAM17]	Recherche par Quadrillage	Positions	EM
[CLMFT14, SFR+15]	Recherche par Quadrillage	Positions	Laser
[VdHOGT21]	Recherche par Quadrillage	Délai d'injection	*Power Glitch*

TABLE 2.3 – Comparaison des travaux précédents en fonction de la technique d'optimisation, l'usage d'une technique de réduction de l'espace de recherche, et de la technique d'injection de faute.

[MAM17] proposent d'acquérir préalablement des traces électromagnétiques, afin de détecter les zones d'intérêt dans le but de réduire l'espace des paramètres, ici les positions x,y de la pointe de la sonde électromagnétique.

2.3.3.2 Délais d'injection

Si les approches précédentes de réduction des positions ciblées ne sont pas généralisables à toutes les techniques d'injection (e.g. *power glitch*), les approches de réduction des délais d'injection le sont. Notamment, il est possible d'isoler la recherche du délai d'injection des paramètres d'équipement. Par exemple, Carpi et al. [CPB+13] divisent le problème d'optimisation en deux étapes ; tout d'abord en optimisant la forme des *power glitches* injectés indépendamment du délai d'injection, puis dans un second temps, en se concentrant uniquement sur le délai d'injection avec les meilleurs *glitches* identifiés. De plus, il est possible d'utiliser des méthodologies d'analyse de vulnérabilités à l'injection de faute dans le but d'identifier les sections critiques de l'application ciblée. Récemment, Van den Herrewegen et al. [VdHOGT21] ont utilisé de l'exécution symbolique [Kin76] pour construire des chemins d'attaque sur le *bootloader* d'un microcontrôleur Renesas 78K0. À partir de ces chemins d'attaque, ils réduisent significativement l'espace de recherche des délais d'injection, ce qui permet de se concentrer sur l'optimisation des paramètres d'équipement.

2.3.4 Comparatif

Le tableau 2.3 résume les travaux, présentés dans cette section, sur les différentes méthodes pour améliorer la recherche des paramètres d'attaque, en fonction de la technique d'injection de faute et de la technique d'optimisation utilisée.

2.4 Caractérisation de fautes

Au même titre que l'optimisation des paramètres d'attaque, la *caractérisation de fautes*, qui consiste à analyser et comprendre les effets des fautes sur le microcontrôleur ciblé, est une tâche complexe. Si cette étape est généralement chronophage et fastidieuse, principalement à cause de la difficulté à observer l'effet induit par la faute. Cette section détaille uniquement les travaux récents et les techniques employées pour mieux observer et comprendre les effets des fautes. La modélisation et la simulation de ces effets seront détaillées par la suite à la section section 2.5.

2.4.1 Principe

La caractérisation de fautes regroupe l'ensemble des méthodes et techniques dont le but est d'identifier et de comprendre les effets des fautes sur un microcontrôleur. Cela consiste dans un premier temps à injecter un grand nombre de fautes de manière à tester différentes combinaisons de paramètres d'équipement, puis à analyser les comportements observés. La complexité réside dans la difficulté à observer les effets produits par la faute injectée, ces derniers pouvant être masqués et donc difficilement observables par l'utilisateur. Plusieurs techniques ont été proposées pour répondre à cette problématique.

2.4.1.1 Test de caractérisation de fautes

Contrairement à l'application cible, un *test de caractérisation de fautes* est un programme conçu spécifiquement pour récupérer un maximum d'informations sur les comportements fautés du microcontrôleur ciblé, de manière à analyser plus facilement les effets des fautes. S'il existe de nombreux tests différents dans la littérature, ils s'articulent généralement autour de trois parties :

- **Prologue** Initialisation de l'état interne du microcontrôleur, qui le plus souvent, se limite à un ensemble de variables V (registres généraux, emplacement mémoires, etc.).
- **Cible** Une séquence d'instruction I où les fautes seront injectées. Dans la littérature, le nombre et le choix des instructions varient d'un test à un autre. Nous reviendrons sur ce point plus en détail au chapitre 5.
- **Épilogue** À la fin de la cible, l'état interne final du microcontrôleur, c'est-à-dire les valeurs finales de V, est sauvegardé.

Noter que les tests de caractérisation de fautes sont généralement instrumentés, au niveau des parties *Prologue* ou *Épilogue*, pour initialiser l'état interne, récupérer l'état final du microcontrôleur ou encore faciliter la synchronisation du délai d'injection.

En comparant l'état final de référence (sans faute) et celui fauté, on peut émettre des hypothèses sur l'effet des fautes sur le microcontrôleur. Cependant, plusieurs hypothèses différentes peuvent souvent expliquer le même résultat fauté ! Par exemple, si la *Cible* du test consiste à additionner les nombres de 1 à 9, soit 45, et que nous obtenons 42 après

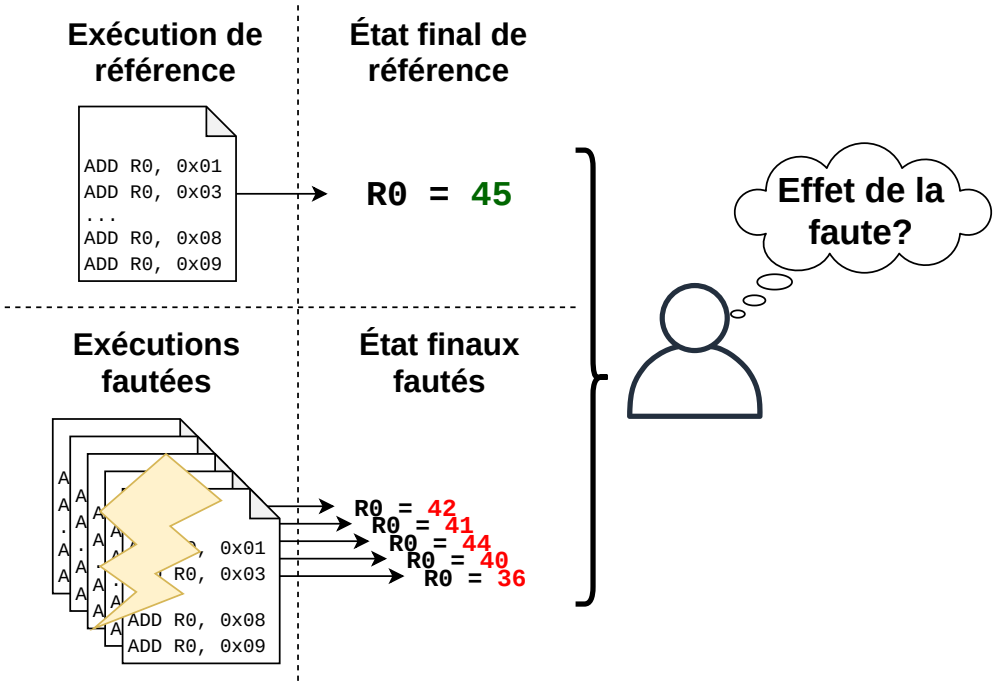


FIGURE 2.12 – Vue schématique du processus de caractérisation de fautes.

l'injection de faute, comment distinguer une faute sur le bus de donnée d'une faute sur le bus d'instruction (figure 2.12) ? Pour cela, il est nécessaire de croiser plusieurs résultats fautés différents avant de confirmer une hypothèse.

Le choix des instructions dans la partie *Cible* dépend de l'effet de faute que l'on souhaite analyser. Par exemple, le tableau 2.4 détaille les instructions I de la *Cible* et l'initialisation des variables V du *Prologue* des tests T1 et T2. Ces tests sont inspirés de travaux récents ([CMD+18, PHM+19, TBC19]) pour mettre en évidence des corruptions d'instruction. Nous reviendrons plus en détail sur ces tests au chapitre 5. En particulier, nous étudierons l'influence des instructions de la partie *Cible* sur la propagation des effets de faute.

Si ces tests, fréquemment utilisés, ont permis de mieux comprendre les effets des fautes sur les microcontrôleurs, ce n'est pas sans limitations. Notamment, cela suppose que les effets des fautes injectées dans la partie *Cible* seront observables à la partie *Épilogue* ce qui n'est pas nécessairement le cas. En particulier, nous verrons dans le chapitre 5 qu'en fonction des choix de conception de la *Cible* (nombre d'instruction, type d'instruction, etc.) certains effets peuvent être masqués. De plus, en cas d'une interruption inopinée du microcontrôleur, il ne sera pas possible d'observer l'état interne de ce dernier. Cependant, il est possible, dans certains cas, de récupérer des informations supplémentaires grâce aux exceptions matérielles.

2.4.1.2 Exception matérielle

Lorsqu'une faute conduit au chargement d'une instruction non définie ou à un accès mémoire non aligné, certaines architectures génèrent des exceptions matérielles. En analysant ces exceptions matérielles, cela permet de récupérer des informations supplémentaires pour

	Test de Caractérisation de fautes T1				Test de Caractérisation de fautes T2			

I	`adds r0, #2`
	`adds r1, #3`
	`adds r2, #5`
	`adds r3, #7`
	`adds r4, #11`
	`adds r5, #13`
	`adds r6, #17`
	`adds r7, #19` } Repeat n times
					`mov r0, r0`
					`mov r1, r1`
					`mov r2, r2`
					`mov r3, r3`
					`mov r4, r4`
					`mov r5, r5`
					`mov r6, r6`
					`mov r7, r7` } Repeat n times			

V	R0	0x00000000	R1	0x11111111	R0	0x00000000	R1	0x11111111
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE 2.4 – Tests de caractérisation de fautes T1 and T2

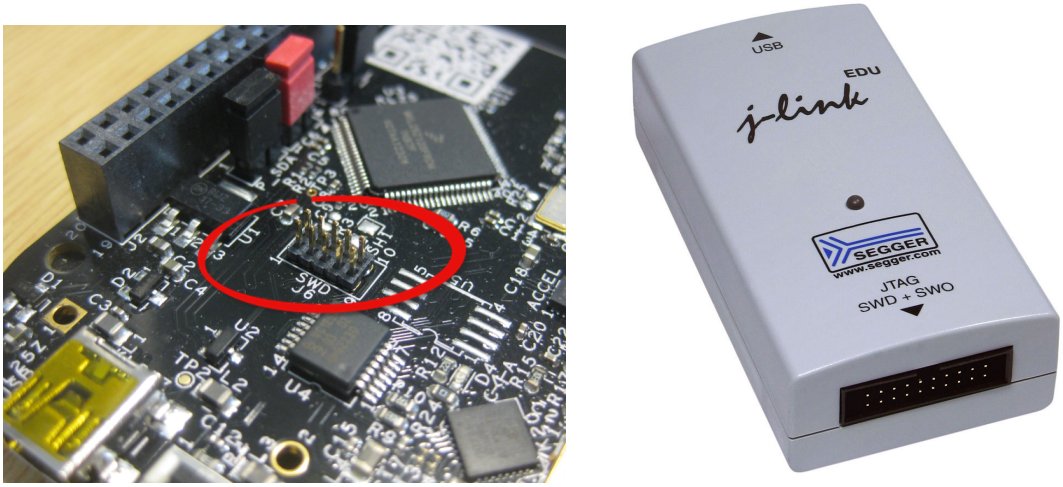
comprendre un peu plus les effets des fautes sur le microcontrôleur cible. Par exemple, l'architecture ARMv7-M peut générer les exceptions suivantes ([Kei17]) :

- L'exception *HardFault* désigne l'exception matérielle par défaut, pouvant être déclenchée à cause d'une erreur pendant le traitement d'une exception.
- *MemManage* détecte les violations d'accès mémoire, comme l'exécution de code depuis une zone mémoire avec accès en lecture/écriture uniquement.
- *BusFault* décèle la présence d'erreurs sur le bus de donnée ou d'instruction, lors de la lecture/écriture de données, ou le chargement d'une instruction.
- *UsageFault* se déclenche lors de l'exécution d'une instruction non définie, d'un accès mémoire non aligné, d'une division par zéro, etc.

Pour chaque exception, les dernières valeurs des 4 premiers registres `R0-R3`, du registre `PC` et du registre `LR` sont sauvegardées et peuvent être utilisées pour analyser l'effet de la faute. En particulier, ces informations sont utilisées par Hummel [Hum14] pour déterminer quelle instruction a été fautée, à ≈ 5 instructions près.

2.4.1.3 On-chip debugger

La plupart des microcontrôleurs intègrent un *On-Chip Debugger* (OCD) pour aider les utilisateurs à déboguer leur code. Un OCD intègre plusieurs fonctionnalités utiles comme écrire et lire en mémoire, modifier les valeurs des registres, charger le code machine de l'application sur le microcontrôleur, ou encore interrompre l'exécution de l'application avec des *breakpoints*. L'OCD s'utilise généralement via un port de débug (e.g. *Serial Wire Debug*, SWD) avec une sonde de débogage (figure 2.13). Cependant l'OCD n'est pas toujours disponible pendant une évaluation de sécurité.



(a) (b)

FIGURE 2.13 – (a) Un port SWD [Sty13] et (b) une sonde de débogage [Seg06].

L'OCD permet d'avoir un contrôle précis sur l'exécution du programme avec les *breakpoints*. Par exemple, Moro et al. [MDH+13] les utilisent pour interrompre l'exécution après une faute pour observer l'état interne du microcontrôleur.

Cependant, il n'est pas toujours facile de choisir où placer les *breakpoints*. En effet, une faute peut sauter le *breakpoint* ou modifier le flot de contrôle. Dans les deux cas, le *breakpoint* ne sera jamais atteint, et l'état interne ne pourra donc pas être observé. De plus, certaines techniques d'injection de faute, notamment les *power glitches*, ne sont pas adaptées à cette approche, car la perturbation peut durer pendant plus d'une dizaine d'instructions, compliquant d'autant plus le positionnement des *breakpoints*.

2.4.1.4 Tracing

Si l'OCD est disponible pour une grande majorité des microcontrôleurs du marché, le *tracing* n'est pas souvent intégré à ces derniers. Le *tracing* est un mécanisme de débogage qui permet de générer une trace d'exécution complète du programme, avec notamment les accès mémoire et les cycles pour chaque instruction exécutée. On retrouve cette fonctionnalité sur certains cœurs ARM (e.g. ARM Cortex-A8) sous le terme de *Embedded Trace Macrocell* (ETM). Le mécanisme de *Tracing* est rarement disponible pendant une évaluation de sécurité.

Hummel [Hum14] propose pour la première fois en 2014 d'utiliser ce mécanisme dans le but d'observer plus facilement les effets des fautes. Malheureusement, le *tracing* ne s'adapte pas facilement aux injections de faute. En effet, l'opcode et l'adresse de l'instruction exécutée n'est pas contenue dans la trace produite et doivent être dérivés à partir du code assembleur. Cela n'est pas un problème pour une utilisation classique de débogage car le code est connu et le flot de contrôle n'est pas altéré. Cependant, lorsqu'une faute modifie le flot de contrôle, ce qui arrive fréquemment, la trace générée n'est pas utilisable.

Travaux Existants	Technique d'Injection	Cible	Technique de caractérisation	Effet des fautes sur la cible
[BGV11]	Clock Glitch	µC 8-bit	Tests	Corruption d'instruction Saut d'instruction
[MDH+13]	EM	µC 32-bit	Tests OCD Exception	Corruption d'instruction
[Hum14]	EM	SoC 32-bit	Tests OCD Exception Tracing	Saut d'instruction Corruption de registre
[KH14]	Clock Glitch	µC 32-bit	Tests	Corruption d'instruction Saut d'instruction Corruption de registre
[RNR+15]	EM	µC 32-bit	Tests OCD	Corruption de cache
[TSW16]	Power Glitch	SoC 32-bit	Tests Exception	Corruption d'instruction
[CMD+18]	Laser	µC 32-bit	Tests	Corruption d'instruction
[PHM+19]	EM	SoC 32-bit	Tests	Corruption d'instruction Saut d'instruction Corruption de registre
[TBC19]	EM	SoC 64-bit	Tests	Corruption d'instruction Corruption de registre
[DRPR19]	Laser	µC 8-bit	Tests	Saut d'instruction
[TBE+21]	EM	SoC 64-bit	Tests OCD	Corruption de cache

TABLE 2.5 – Comparaison des récents travaux de caractérisation de fautes sur différents microcontrôleurs (µC) et System-on-Chip (SoC).

2.4.2 Travaux récents

La caractérisation des fautes sur les microcontrôleurs est un domaine actif depuis les premiers travaux de Balasch et al. en 2011 [BGV11]. Ces derniers étudient les effets des fautes induites par des *clock glitches* sur un microcontrôleur 8-bit, avec plusieurs tests de caractérisation de fautes différents. Ils observent majoritairement des remplacements ou des corruptions d'instruction en faisant varier la durée de la perturbation du signal d'horloge. Ce premier constat sera confirmé par Korak et al. [KH14], cette fois-ci sur un microcontrôleur 32-bit.

Depuis, d'autres techniques d'injection de faute ont été testées (*Power glitch*, EM et Laser) sur une grande variété d'architectures différentes, des microcontrôleurs 8-bit jusqu'aux processeurs multi-cœur 64-bit. Le tableau 2.5 présente les travaux récents de caractérisation

de fautes sur différentes architectures. On peut remarquer que si les tests de caractérisation sont très populaires pour observer les effets des fautes, l'utilisation d'exceptions matérielles ([MDH+13, Hum14, TSW16]), de l'OCD ([MDH+13, Hum14, RNR+15, TBE+21]) ou encore du tracing ([Hum14] est moins fréquente.

Enfin, bien que les effets des fautes varient selon la technique d'injection de fautes considérée, les tests utilisés et la cible évaluée, les travaux récents décrivent majoritairement des effets de corruption d'instruction ([BGV11, MDH+13, KH14, TSW16, CMD+18, PHM+19, TBC19]) ou de saut d'instructions ([BGV11, Hum14, KH14, PHM+19, DRPR19]). Dans la section suivante, nous détaillerons la modélisation et la simulation des effets de faute.

2.5 Modélisation et simulation des fautes

Les effets des fautes observés sont souvent modélisés pour étudier plus facilement la robustesse d'une application (un bootloader, un algorithme cryptographique, etc.) aux fautes. De manière générale, les modèles de faute sont utilisés pour évaluer la vulnérabilité à l'injection de faute via des outils de simulation de faute. Ces outils permettent d'analyser la robustesse d'une application vis-à-vis des modèles de faute établis, automatiquement, à moindre coût, et sans risque de détérioration physique du circuit. Différents niveaux d'abstraction peuvent être utilisés pour modéliser une faute en fonction du niveau d'analyse souhaité. Après avoir précisé ces niveaux, nous détaillerons les approches visant à combler l'écart entre ces niveaux d'analyse. Nous terminerons sur la problématique des fautes multiples.

2.5.1 Niveau d'abstraction

Les effets d'une faute peuvent être analysés à différents niveaux d'abstraction, comme illustré à la figure 2.14. Chaque niveau d'analyse est un compromis entre *fidélité*, *extensibilité* et *praticité* (figure 2.15). La *fidélité* désigne la représentativité de l'effet décrit par le modèle. L'*extensibilité* correspond à la capacité à s'adapter à l'évaluation d'applications plus complexes, c'est-à-dire à passer à l'échelle. Enfin la *praticité* représente la facilité d'utilisation dans le contexte d'évaluation de sécurité.

2.5.1.1 Niveau Circuit

Pour étudier le comportement des transistors soumis à des perturbations physiques, les effets des fautes peuvent être modélisés au *niveau circuit* (figure 2.16a) avec des simulateurs de circuits électroniques (SPICE, Eldo, etc.). Ces travaux sont intéressants pour confirmer ou réfuter des hypothèses de modèles à plus haut niveau, en particulier, les trois modèles suivant au niveau logique :

- **Bit-set** la valeur d'un bit est forcée à 1.
- **Bit-reset** la valeur d'un bit est forcée à 0.
- **Bit-flip** la valeur d'un bit est inversée.

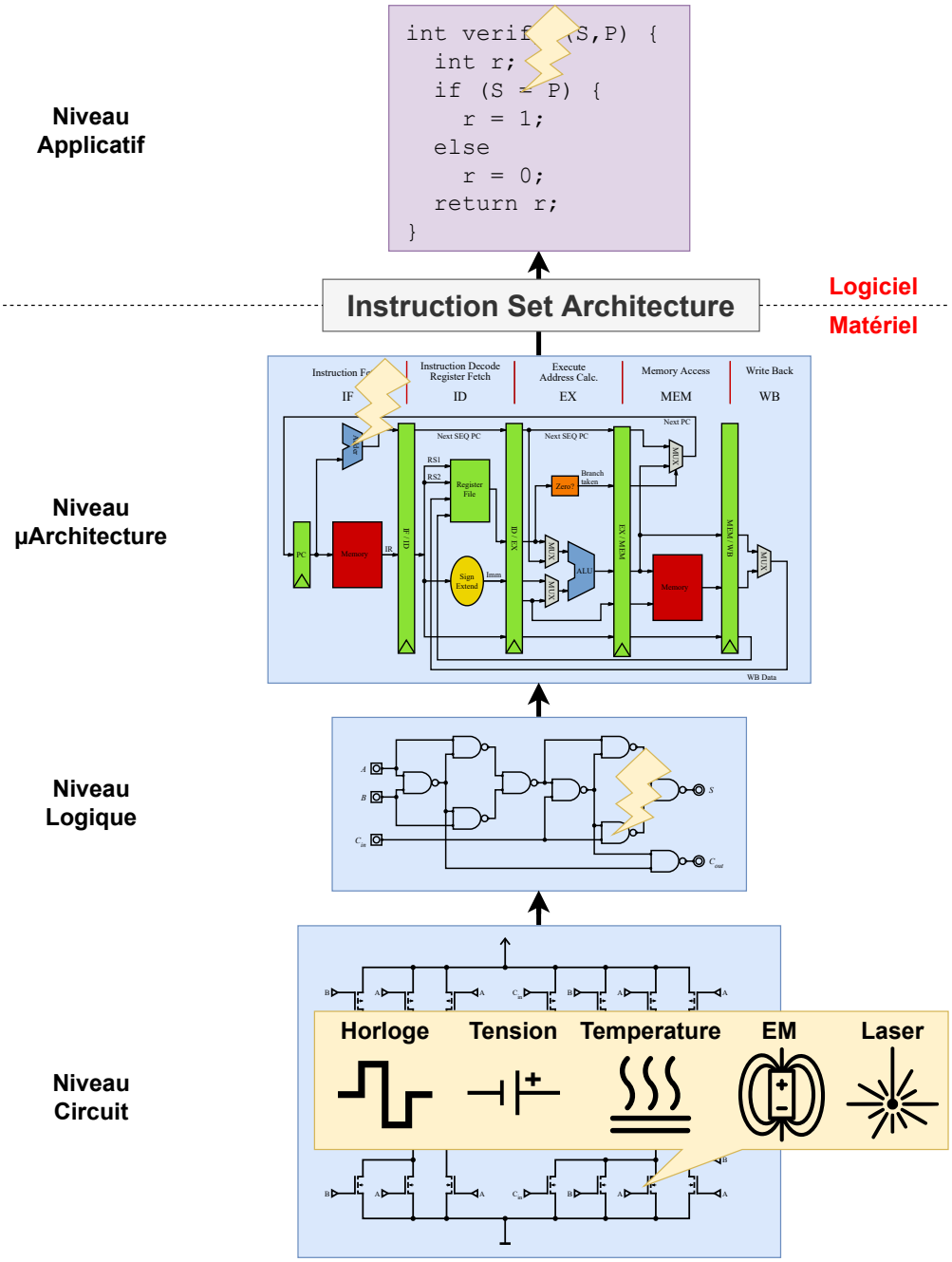


FIGURE 2.14 – Illustration du cycle de vie d'une faute [YSW18]. Au *niveau circuit*, une perturbation physique fait commuter l'état d'un ou plusieurs transistors. Ces transistors fautés, au *niveau logique*, font partie d'un additionneur qui génère, à son tour, un bit-set sur le résultat de l'addition. Au *niveau de la microarchitecture*, le compteur ordinal, mis-à-jour par l'additionneur fauté, pointe alors vers la mauvaise instruction. Le chargement et l'exécution de la mauvaise instruction, au *niveau ISA*, conduit à ne pas mettre à jour le registre d'état. Finalement, au *niveau applicatif*, le test de vérification est inversé et l'utilisateur est authentifié sans connaissance du secret.

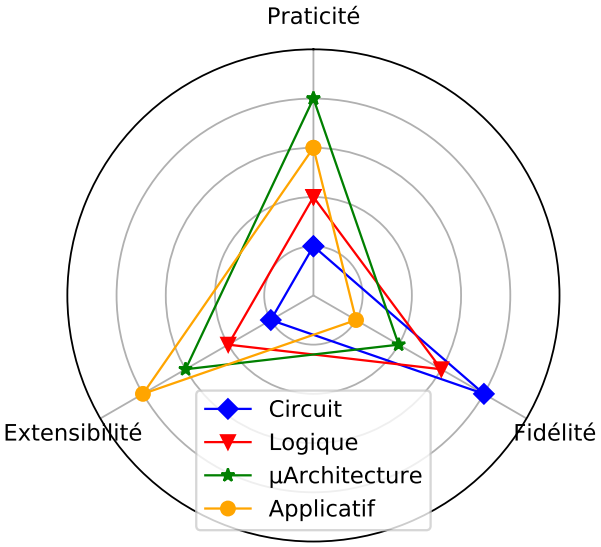


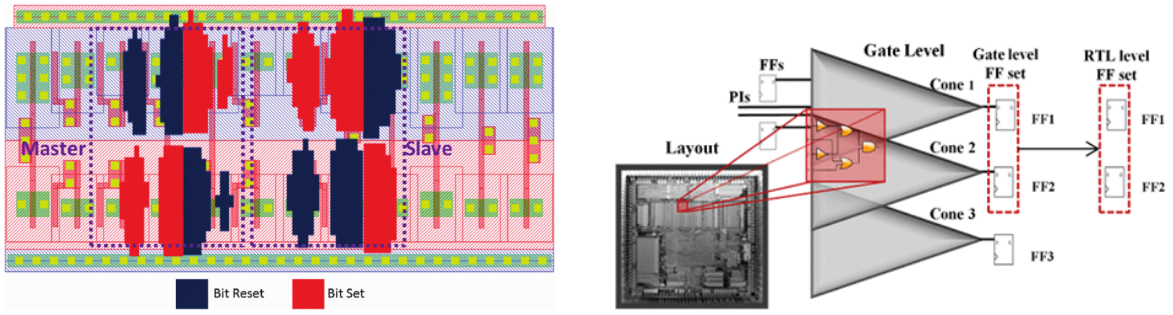
FIGURE 2.15 – Comparaison des niveaux d'analyse en terme de vitesse, précision et facilité d'utilisation.

Par exemple, les travaux de Roscian et al. [RSDT13] sur l'effet des perturbations par lumière focalisée confirment les modèles de bit-set et bit-reset sur des cellules mémoires SRAM (finesse de gravure 0.25 µm) mais réfutent le modèle bit-flip. Cependant, en fonction de la finesse de gravure, le modèle de bit-flip n'est pas à écarter totalement, comme le montre Dutertre et al. [DBC+18] sur une technologie de 28 nm. Plus récemment, les travaux de Dumont et al. [DLM20] théorisent le modèle d'échantillonnage, propre aux perturbations électromagnétiques. D'après ce modèle, les ondes EM introduisent une perturbation de la tension d'alimentation, conduisant les bascules du circuit à échantillonner la mauvaise valeur, confirmant les modèles de bit-set, bit-reset et bit-flip au niveau logique.

2.5.1.2 Niveau Logique

Malgré la fidélité des modèles de faute au niveau circuit, le niveau d'analyse circuit reste limité en termes de praticité et d'extensibilité et, par conséquent, n'est jamais utilisé dans le contexte d'évaluation de sécurité.

Une analyse au niveau logique permet de passer à l'échelle tout en conservant une fidélité de simulation proche du niveau circuit. Au lieu de s'intéresser aux interactions physiques entre les transistors du circuit, l'accent est mis sur les transferts de données et les opérations logiques entre les bascules du circuit. En particulier, Papadimitriou et al. [PTH+15] proposent une méthodologie pour extraire les bascules localement touchées par une perturbation laser (figure 2.16b). Simuler des bit-set, bit-reset et bit-flip au niveau des bascules permet de mettre en évidence de nouveaux modèles au niveau microarchitecture [LBD+18].



(a) (b)

FIGURE 2.16 – (a) Modélisation au niveau transistor de bit-set et bit-reset par perturbation laser [CBD+15] (b) Extraction et analyse des bascules fautées [PTH+15]

2.5.1.3 Niveau Microarchitecture, Niveau ISA

Pour analyser au niveau logique, il est nécessaire d'avoir connaissance du modèle RTL (*Register-Transfer Level*) de l'architecture ciblée, qui n'est généralement pas disponible pour une grande majorité des évaluations de sécurité. Ainsi l'analyse au niveau de la microarchitecture ou au niveau du jeu d'instruction (*Instruction Set Architecture*, ISA) est souvent favorisée, car ne nécessitant que 1) le manuel de référence d'architecture associé au microcontrôleur ciblé et 2) le binaire de l'application évaluée. En comparaison avec le niveau logique, ces niveaux permettent de passer à échelle sur des applications plus complexes, au détriment de la fidélité des fautes simulées.

L'analyse au niveau microarchitecture étudie l'impact des fautes sur le cycle d'exécution d'une instruction (chargement, décodage, exécution) et les interactions entre les registres, la mémoire et les unités arithmétique et logique. C'est le niveau d'analyse le plus populaire dans la littérature scientifique et il existe une grande variété de modèles de faute. Les effets des fautes peuvent être modélisés sous la forme de bit-set, bit-reset et bit-flip au niveau des registres ou des données mémoire [Hum14, KH14, Dur16, TBC19, PHM+19] mais sur l'opcode de l'instruction chargée [CMD+18]. Les modèles de corruption d'instruction sont d'ailleurs utilisés par Timmers et al. [TSW16] pour contrôler la valeur du registre PC (compteur ordinal) dans le but d'exécuter du code arbitraire. Si ces modèles sont souvent exprimés au niveau ISA plutôt qu'au niveau de la microarchitecture à proprement parler, certains modèles plus complexes prennent en compte les éléments micro-architecturaux comme les caches [RNR+15, TBE+21]. Enfin, le modèle de saut d'instruction revient régulièrement dans la littérature. En particulier, Dutertre et al. ont montré qu'il était possible d'induire des sauts d'instruction de longueur arbitraire en fonction de la durée de la pulsation laser [DRPR19].

2.5.1.4 Niveau Applicatif

L'analyse au niveau applicatif permet de couvrir plus facilement des applications complexes mais requiert un accès au code source de l'application ciblée, ce qui est néanmoins souvent le cas en évaluation de sécurité. Le principal défaut de ce niveau d'analyse reste la fidélité du modèle, les effets modélisés pouvant être différents des effets des fautes injectées

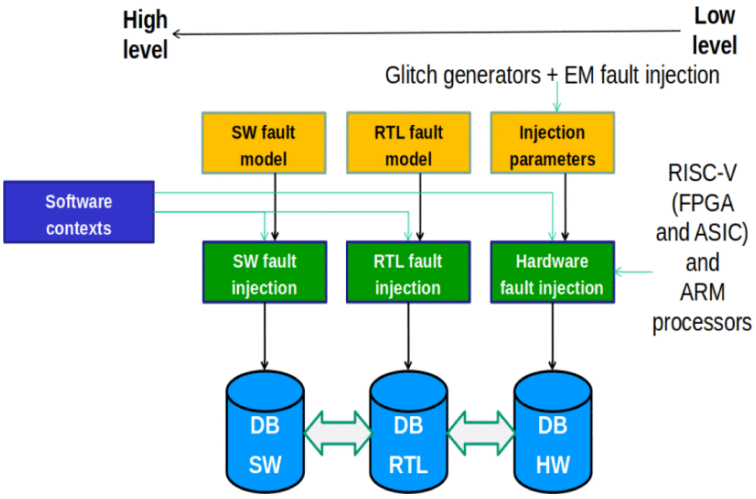


FIGURE 2.17 – Approche proposée par Alshaer et al. [ACD+21]

expérimentalement. Cependant, des méthodologies ont été proposées pour améliorer ce point (sous-section 2.5.2). Les effets des fautes sont modélisés sous la forme de bit-set, bit-reset ou bit-flip sur les variables manipulées par le programme, ou sous la forme d'inversion de test [PMPD14, BEMP20]. Les sauts d'instruction peuvent être aussi simulés au niveau applicatif avec, par exemple, l'instruction `goto` [LHB14].

2.5.2 Combler l'écart

S'il existe de nombreux outils de simulation d'injection de faute dans la littérature, nous nous intéresserons seulement ici aux outils et approches pour combler l'écart soit 1) entre les effets des fautes simulées et les effets fautes injectées expérimentalement ou 2) entre les différents niveaux d'analyse présentés précédemment.

2.5.2.1 Entre simulation et expérimentation

Dureuil et al. [DPdC+15] proposent une approche pour inférer des modèles de fautes ISA à partir des résultats d'une caractérisation de faute. Les modèles de faute sont probabilistes et prennent en compte la probabilité d'obtenir chaque effet observé expérimentalement. À partir des modèles inférés, l'évaluation de sécurité est conduite au niveau ISA avec l'outil `CELTIC` (chapitre 3), et permet d'identifier les sections critiques de l'application ainsi que d'aider à la notation du potentiel d'attaque [Joi20]. Le fait que les modèles soient tirés d'expérimentations augmente la fidélité des effets injectés pendant la simulation. Similairement, Given et al. [GWJL18] vérifient la fidélité des modèles simulés au niveau ISA avec des résultats expérimentaux pour identifier les modèles les plus représentatifs des fautes observées. Cependant, alors que Dureuil et al. évaluent l'application en fonction des résultats expérimentaux, Given et al. utilisent les résultats expérimentaux pour vérifier la pertinence des résultats simulés.

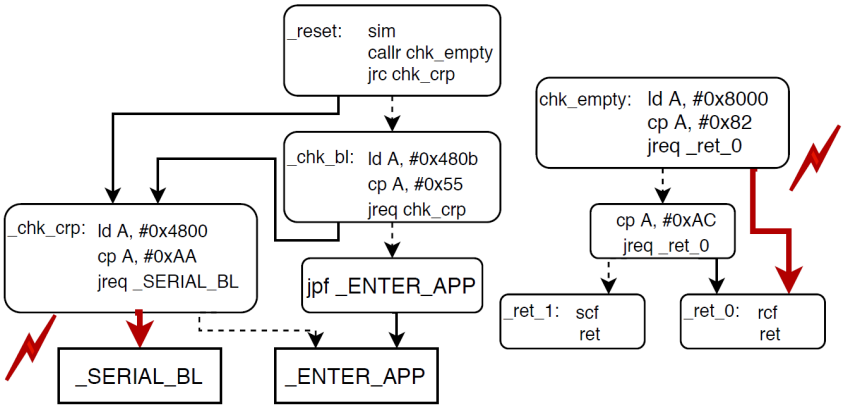


FIGURE 2.18 – Le bootloader peut être réactivé avec deux fautes : la première permet de tromper le programme en faisant croire que la mémoire Flash utilisateur est vide (`chk_empty`), la deuxième contourne le mécanisme de CRP (`chk_crp`) [VdHOGT21].

2.5.2.2 Entre niveau d'analyse

Riviere et al. [RPL+14] combinent une approche au niveau applicatif avec une approche au niveau microarchitecture. Au niveau applicatif, l'outil `LAZART` [PMPD14] simule des injections de fautes multiples avec un modèle d'inversion de test ce qui permet, au niveau microarchitecture, de réduire significativement le nombre de fautes à injecter avec l'outil `EFS` [BBC+14], en ciblant uniquement les sections critiques de l'application identifiées par `LAZART`.

Laurent et al [LDPPB21] analysent les fautes injectées au niveau logique pour identifier les modèles au niveau applicatif les plus pertinents. Une fois les modèles au niveau applicatif validés, ils peuvent être ensuite utilisés pour évaluer l'application ciblée avec des outils d'analyse statique comme `FRAMA-C` [CKK+12] ou dynamique comme `LAZART`. À noter que les auteurs soulignent que de nombreux modèles de faute au niveau applicatif sont nécessaires pour atteindre une bonne couverture des effets observés au niveau logique.

Enfin, Alshaer et al. [ACD+21] complètent l'approche de Laurent et al. en combinant les résultats expérimentaux avec les résultats d'analyse au niveau applicatif et logique (figure 2.17) ce qui permet de modéliser plus facilement les effets des fautes observés expérimentalement.

2.5.3 Fautes multiples

Depuis la première attaque par faute multiple proposée par Kim et al. [KQ07], peu de travaux ont été proposés pour améliorer l'identification et l'exploitation de vulnérabilité à l'injection de faute multiple. Pourtant l'injection de faute multiple permet de considérablement augmenter le nombre de chemins d'attaque possibles pour contourner un mécanisme de sécurité. En particulier, Van den Herrewegen et al. [VdHOGT21] ont réussi à modifier le flot de contrôle à deux reprises avec des *power glitches* pour réactiver le bootloader d'un microcontrôleur 8-bit (figure 2.18). Cet exemple est la première attaque par fautes mul-

tiples documentée sur une cible réelle, illustrant l'importance de prendre en considération ces dernières pendant les évaluations de sécurité.

Cependant, l'analyse de la vulnérabilité à l'injection de faute multiple est une tâche complexe en raison de l'explosion combinatoire dû au nombre de fautes injectées, augmentant significativement le temps consacré à l'analyse de l'application pour identifier les chemins d'attaques les plus prometteurs.

À ce jour, relativement peu d'outils d'analyse prennent en compte l'injection de fautes multiples nativement. On pourra noter parmi eux l'outil `LAZART` qui se base sur de la coloration de graphe couplée à de l'exécution concolique pour identifier les chemins possibles pour atteindre un bloc visé avec des modèles au niveau applicatif comme de l'inversion de test. Du point de vue des développeurs, cet outil permet notamment d'optimiser les contre-mesures d'une application en détectant celles inutiles [BEMP20].

2.6 Conclusion

Dans ce chapitre nous avons présenté, non seulement les différentes techniques pour injecter des fautes par perturbation sur un microcontrôleur, mais surtout les approches pour identifier les paramètres d'attaque et comprendre les effets des fautes sur la cible. Ces deux aspects sont très importants en injection de faute. Il peut être facile d'oublier l'aspect physique de ce type d'attaque et ne se focaliser que sur la partie analyse de code, sans prendre en considération les spécificités de la cible.

Ainsi, la première problématique est de réussir à lier les résultats simulés avec la réalité physique des paramètres d'équipement. L'étape consistant à passer de la simulation à l'expérimentation est rarement abordée dans la littérature, et pourtant, cette étape est particulièrement complexe et chronophage, du fait de la taille de l'espace des paramètres d'équipement. Ce point peut devenir bloquant lors de l'analyse de la vulnérabilité aux fautes multiples, du fait de l'explosion combinatoire dû au nombre de fautes injectées, complexifiant d'une part la simulation d'injection de faute et d'autre part l'expérimentation. Pourtant, les attaques par fautes multiples risquent de devenir de plus en plus fréquentes, particulièrement depuis la publication d'une attaque double faute entièrement documentée sur le bootloader d'un microcontrôleur 8-bit (figure 2.18). Il est donc important de les prendre en compte, et réussir à comprendre et identifier plus facilement cette nouvelle menace.

L'autre point important qui mérite d'être développé concerne les tests de caractérisation de fautes. Il en existe une multitude dans la littérature, mais il n'y a pas de métrique de performance pour comparer l'efficacité de ces derniers pour identifier les effets des fautes. Par exemple, en fonction des instructions choisies dans la partie *Cible* du test, certains effets ne seront pas observables. Cela peut conduire à une mauvaise interprétation de la sensibilité aux fautes du microcontrôleur ciblé, ce qui peut avoir des conséquences sur l'évaluation de sécurité.

Enfin, les techniques d'optimisation utilisées dans le contexte d'injection de faute ne suivent pas l'état de l'art des techniques d'optimisation utilisées dans d'autres domaines, comme l'apprentissage automatique. Il serait intéressant de confronter les techniques récentes d'optimisation aux problématiques spécifiques de l'injection de faute. Pour résumer, cette

thèse vise à :

- Proposer une méthodologie d'identification et d'exploitation des vulnérabilités à l'injection de faute multiple. Le but est de réussir à lier les résultats simulés aux paramètres d'attaque dans le contexte d'injection de faute multiple (chapitre 4).
- Formaliser les propriétés des tests de caractérisation de la littérature dans le but d'améliorer la conception de ces derniers (chapitre 5).
- Etudier de nouvelles techniques d'optimisation jamais utilisées dans le contexte d'injection de faute pour accélérer l'identification des paramètres d'attaques (chapitre 6).

Avant de détailler nos travaux, nous présentons, au chapitre suivant, deux outils importants, `CELTIC` et la `GLITCH STATION`, qui ont été développés pendant cette thèse pour simuler et injecter des fautes, respectivement.

Chapitre 3

Outillage pour l'injection de faute

Sommaire

3.1 Introduction

Dans ce chapitre, nous présentons les outils qui seront utilisés dans les chapitres suivants pour simuler ou injecter des fautes.

D'une part, on détaillera le fonctionnement de `CELTIC`, qui est un simulateur d'injection de faute développé initialement par Dureuil [Dur16], et qui sera utilisé pour identifier des attaques par fautes multiples au chapitre 4, et pour analyser l'efficacité des tests de caractérisation de fautes au chapitre 5.

D'autre part, on présentera un nouvel équipement d'injection de faute par perturbation de la tension d'alimentation, la `GLITHSTATION`. Cet équipement d'injection de *power glitch* a été développé pendant cette thèse pour être un outil facile d'utilisation, tout-en-un et abordable ($\approx$ 100 €). La particularité de cet outil est d'intégrer un convertisseur numérique-analogique (*Digital-to-Analog Converter*, DAC) pour générer des formes de *glitch* complexes, permettant d'induire des fautes sur une grande variété de microcontrôleurs. Cet outil sera utilisé pour injecter des fautes multiples au chapitre 4, évaluer des tests de caractérisation de fautes au chapitre 5 et contourner les mécanismes de protection de code du *bootloader* du STM32F103RE au chapitre 6.

Ainsi la suite du chapitre s'organise comme suit. À la section 3.2, nous présentons les motivations derrières l'outil `CELTIC` mais aussi l'architecture de ce dernier et les modèles de

faute pris en compte. Ensuite, à la section 2.3 nous détaillons la `GLITHSTATION`, sa conception, son fonctionnement et ses spécificités par rapport aux autres outils existants.

3.2 CELTIC

Afin d'aider l'évaluateur à l'analyse des vulnérabilités à l'injection de faute, des outils ont été développés pour simuler les effets des fautes sur le comportement du microcontrôleur ciblé, dans le but de détecter de potentielles faiblesses d'implémentation. En particulier, le CESTI CEA-Leti utilise, entre autres, un outil développé par Louis Dureuil, `CELTIC` [Dur16].

3.2.1 Motivations

Dureuil relève les contraintes liées à l'analyse de vulnérabilités à l'injection de faute, qui ont conduit au développement de `CELTIC`. Dans le cadre d'une évaluation de sécurité, l'outil idéal doit être :

- **Non-intrusif** Afin de couvrir un maximum de scénarios, il est préférable que l'analyse soit directement effectuée sur le binaire qui a été remis par le client, et donc les approches par instrumentation ou par recompilation ne sont pas adaptées.
- **Multi-architecture** Toujours dans un souci de couvrir un maximum d'évaluation de sécurité, l'outil doit pouvoir supporter plusieurs architectures différentes et être suffisamment flexible de manière à pouvoir en rajouter de nouvelles.
- **Représentatif** L'outil doit pouvoir simuler des injections de faute fidèles à celles obtenues expérimentalement sur le microcontrôleur ciblé. Pour cela, l'analyse doit tenir compte du jeu d'instruction de la cible pour prendre en compte les fautes qui dépendent de l'encodage des instructions.
- **Pertinent** Il faut pouvoir classer les résultats de la simulation selon la criticité des fautes obtenues pour aider l'évaluateur dans la notation du potentiel d'attaque.
- **Robuste** L'outil se destine à être utilisé sur des cas réels d'évaluation. Il doit être suffisamment fiable et robuste pour permettre le passage à l'échelle.

`CELTIC` est l'outil développé en réponse à ces besoins. Les caractéristiques principales de `CELTIC` sont les suivantes :

- **Analyse dynamique au niveau ISA** Pour avoir des résultats proches de la réalité et tenir compte de l'architecture du microcontrôleur cible, le choix de l'analyse dynamique au niveau ISA a été retenu. Plus précisément, `CELTIC` émule l'architecture du microcontrôleur ciblé puis simule les injections de faute selon des modèles au niveau ISA pendant l'exécution du binaire évalué. Comme nous l'avons vu au chapitre 2, ce niveau d'analyse est un bon compromis entre fidélité, praticité et extensibilité.

- **Langage de spécification d'architecture** Pour répondre au besoin de polyvalence et de portabilité, un langage de spécification d'architecture, `GISL`, a été développé, permettant de supporter plusieurs architectures différentes, à partir du moment où ces dernières sont décrites en `GISL`.
- **API Python** L'évaluateur peut utiliser `CELTIC` via une API `PYTHON` [VRD+00] ce qui permet de faciliter la prise en main de l'outil, mais aussi d'utiliser des librairies comme `PANDAS` [M+11], facilitant l'analyse des résultats.
- **Multi-thread et pré-compilation** Enfin, `CELTIC` accélère la simulation d'injection de fautes en répartissant l'ensemble des fautes à injecter entre plusieurs exécutions parallèles. Aussi, `CELTIC` pré-compile le jeu d'instructions du microcontrôleur ciblé en instructions exécutables par la machine hôte, à l'aide du moteur `LLVM MCJIT` [LA04]. Réduire les temps de simulation est important, particulièrement dans le cas de l'injection de fautes multiples.

3.2.2 Contributions

L'outil de simulation d'injection de fautes, `CELTIC`, a fait l'objet d'une refonte complète. Une première raison est purement pratique. L'auteur original de l'outil, n'étant plus au CESTI, le code de l'outil, de plusieurs dizaines de milliers de lignes, n'était plus maintenu. Avec les années, certaines portions de code étaient en doublon ou non terminées, tandis que d'autres étaient difficilement modifiables sans changer l'architecture de l'outil. Ainsi, l'architecture a été repensée et certaines fonctionnalités ont été rajoutées, notamment : la parallélisation des attaques et la pré-compilation des instructions `GISL` en code machine pour réduire le temps de simulation. Les modèles de fautes ont été repensés, il est possible de reproduire tous les modèles de fautes génériques au niveau ISA, mais aussi des modèles de fautes plus complexes (détaillés à la sous-section 3.2.4). De plus, une interface `Python` a été développée pour faciliter l'utilisation de `CELTIC`.

3.2.3 Architecture

Dans les sections suivantes, nous détaillerons brièvement les étapes clés de `CELTIC` à savoir l'initialisation et la compilation du fichier de description d'architecture `GISL`, l'exécution du binaire évalué et la simulation d'injection de fautes. L'architecture générale est détaillée dans la figure 3.1.

3.2.3.1 Initialisation et compilation

La première étape lors de l'initialisation d'un nouveau projet avec `CELTIC` est de compiler le fichier de description d'architecture `GISL` (extension `*.man`). Ce fichier contient les informations nécessaires pour initialiser l'architecture qui sera simulée comme le nombre de registres, la taille mémoire et surtout le jeu d'instructions de l'architecture. La rédaction du fichier *.man* est à la charge de l'évaluateur si l'architecture n'est pas encore décrite en `GISL`. Après les analyses lexicale, syntaxique et sémantique du fichier, chaque instruction décrite

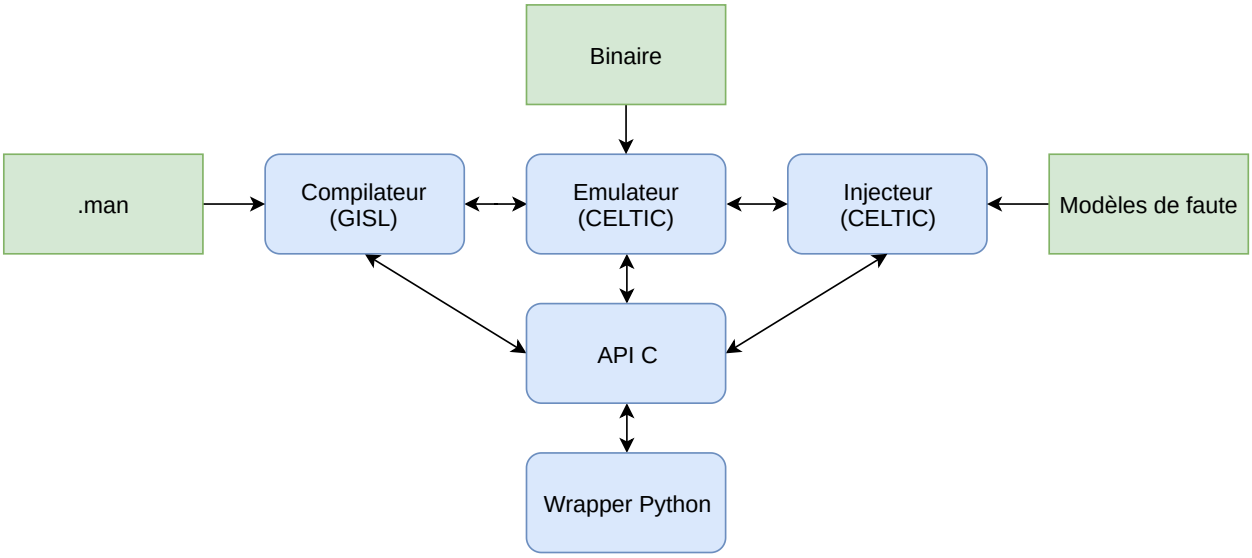


FIGURE 3.1 – Vue globale de l'architecture de `CELTIC`.

dans le jeu d'instruction est traduite en représentation intermédiaire `LLVM` qui sera optimisée puis transformée en code machine exécutable sur la machine hôte. Ainsi, à la fin de cette étape, toutes les instructions du jeu d'instruction du microcontrôleur ciblé sont pré-compilées ce qui permet d'accélérer la simulation d'injection de faute. Le processus de compilation du fichier `GISL` est détaillé à la figure figure 3.2.

Les outils `Flex` et `Bison` [ALSU20] sont utilisés en compilation pour générer des analyseurs lexicaux et syntaxiques à partir d'un fichier décrivant d'une part les expressions rationnelles du langage et d'autre part les règles de grammaire du langage. Le projet `LLVM` [LA04] est une collection d'outils et de librairies modulables et réutilisables permettant de concevoir des compilateurs. Historiquement, `LLVM` est l'acronyme de *Low Level Virtual Machine* mais désigne maintenant le projet `LLVM` dans son ensemble : la représentation intermédiaire `LLVM`, la suite de compilation `Clang`, etc. Après la génération d'un arbre de syntaxe abstraite avec `Flex` et `Bison`, qui est une représentation fortement liée au langage source utilisé, l'arbre de syntaxe abstraite est transformé en une représentation intermédiaire, celle de `LLVM`, agnostique du langage source. Ensuite, la représentation intermédiaire `LLVM` générée est optimisée avec les optimisations de code mise à disposition par `LLVM`. Enfin, le code machine est généré directement à partir de cette représentation intermédiaire.

3.2.3.2 Exécution du binaire

Une fois l'architecture initialisée, `CELTIC` exécute le binaire en suivant le processus détaillé à la figure figure 3.3. Ce processus, en trois étapes, consiste à charger, décoder puis exécuter l'instruction pointée par le compteur ordinal, de manière à mettre à jour l'état courant.

L'état est une structure contenant les valeurs instantanées des registres généraux et des registres spéciaux comme le registre PC (compteur ordinal) ou le registre d'état. L'état contient aussi les valeurs lues et écrites en mémoire jusqu'à lors. Il est possible d'ajouter

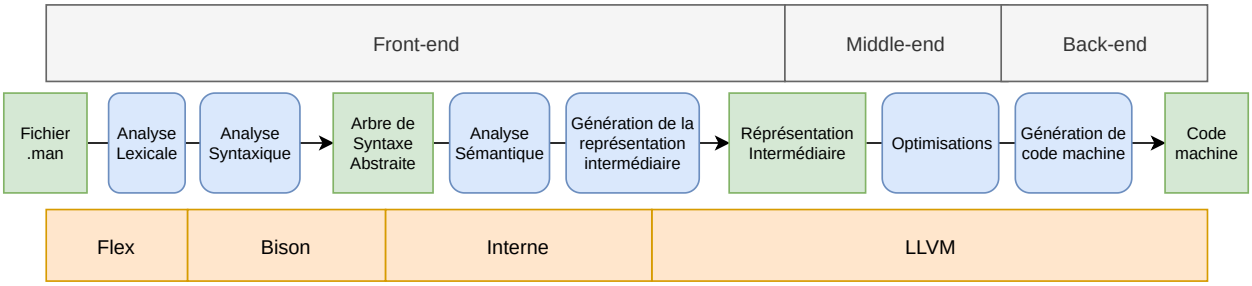


FIGURE 3.2 – Compilation du fichier de description d'architecture GISL.

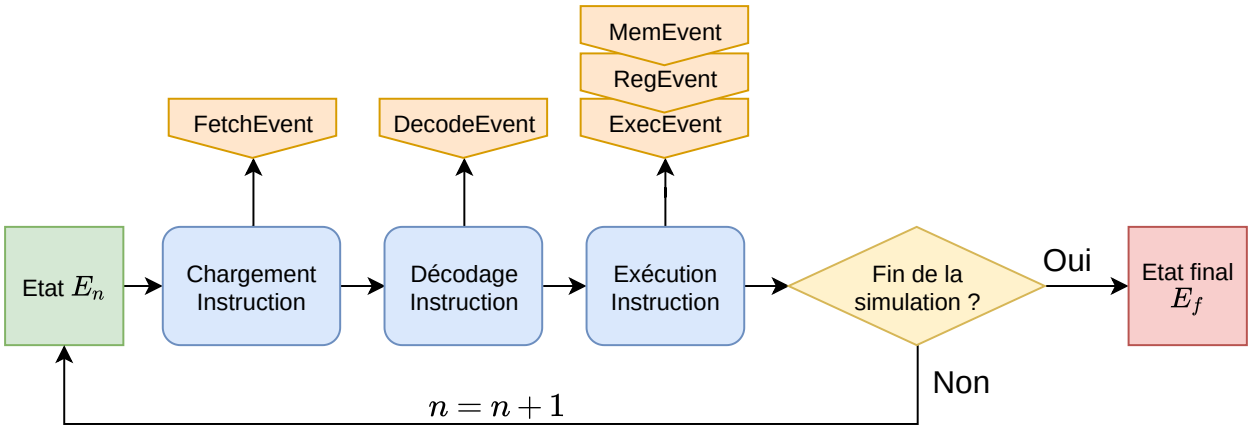


FIGURE 3.3 – Processus d'exécution du binaire évalué avec `CELTIC`.

des *watch variables* pour observer et suivre plus facilement la valeur de certaines variables en mémoire. Par exemple, cela permet de vérifier la valeur de la variable `g_authenticated` pour savoir si l'authentification est acceptée, comme illustré à la figure 3.4.

L'exécution se poursuit jusqu'à atteindre une adresse spécifiée par l'utilisateur ou jusqu'à déclencher une interruption, due par exemple à l'exécution d'une instruction non reconnue.

Enfin, des évènements sont générés à chaque étape de l'exécution du binaire :

- `FetchEvent` est généré pendant le chargement d'instruction et contient l'adresse de l'instruction chargée.
- `DecodeEvent` est généré pendant le décodage de l'instruction et contient l'opcode de l'instruction decodée.
- `ExecEvent` est généré pendant l'exécution de l'instruction et contient l'adresse, l'opcode de l'instruction et diverses informations supplémentaires comme un pointeur vers la description GISL de l'instruction exécutée.
- `RegEvent` est généré pendant la lecture et l'écriture d'un registre mémoire.
- `MemEvent` est généré pendant la lecture et l'écriture d'une donnée en mémoire.

```
print(verifypin_entry)
```

```
PC = 0x80002d4
cycle = 280
R[0] = 0x20000000
R[1] = 0x4
R[2] = 0x2a
R[3] = 0x4
R[4] = 0x0
R[5] = 0x0
R[6] = 0x0
R[7] = 0x20017ff8
R[8] = 0x0
R[9] = 0x0
R[10] = 0x0
R[11] = 0x0
R[12] = 0x0
R[13] = 0x20017ff8
R[14] = 0x8000265
R[15] = 0x0
R[16] = 0x0
R[17] = 0x0
At 0x20000024, g_cardPin = {0x01,0x02,0x03,0x04}
At 0x20000020, g_userPin = {0x2a,0x2a,0x2a,0x2a}
At 0x2000001e, g_authenticated = {0x55}
At 0x2000001c, g_countermeasure = {0x00}
At 0x2000001d, g_ptc = {0x03}
```

FIGURE 3.4 – Affichage de l'état courant dans `CELTIC`.

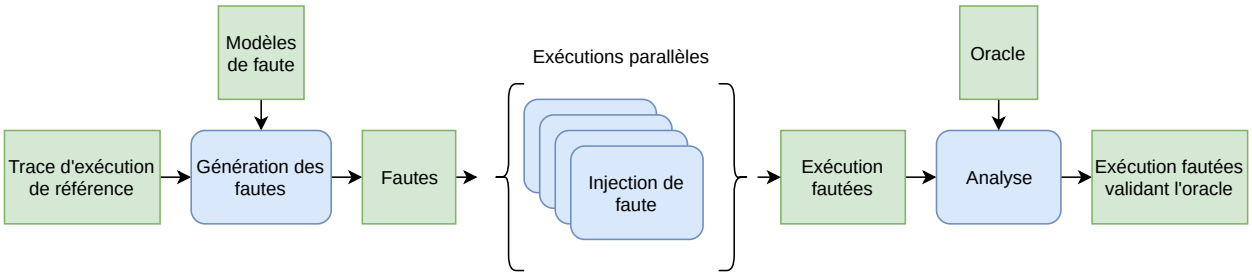


FIGURE 3.5 – Processus de simulation d'injection de faute dans `CELTIC`.

Ces évènements sont ensuite enregistrés pour produire une trace d'exécution. Cette dernière est une liste contenant les états exécutés et les événements générés. Cette trace est utilisée pendant la simulation d'injection de faute.

3.2.3.3 Simulation d'injection de faute

Avec la trace d'exécution de référence, c'est-à-dire sans faute, on détermine les sites d'injection possibles en fonction des modèles de faute considérés. Par exemple, pour pouvoir appliquer un modèle de faute sur les valeurs lues en mémoire, il faut avoir au préalable enregistré un événement `MemEvent`. Une fois les sites d'injection identifiés, on génère les fautes qui seront ensuite injectées pendant l'exécution du binaire. En particulier, c'est cette étape d'injection de faute qui est parallélisée dans `CELTIC`, comme illustré à la figure 3.5.

Enfin, avec l'oracle choisi par l'utilisateur, qui est une expression booléenne statuant sur la réussite de l'attaque (par exemple, "l'utilisateur est authentifié" peut se traduire par l'expression booléenne `g_authenticated == 0xAA`), on analyse les traces d'exécution fautées obtenues pour finalement ne conserver que celles permettant de valider l'oracle. Nous verrons

Catégorie	Famille de modèles de faute	Description
Corruption d'Instruction	F0	Corruption du cache d'instruction
	F1	Saut d'instructions
	F2	Bit-sets sur l'instruction décodée
	F3	Bit-resets sur l'instruction décodée
Corruption de Registre	F4	Bit-sets sur la valeur du registre lue
	F5	Bit-resets sur la valeur du registre lue
Corruption de mémoire	F6	Bit-sets sur la valeur de la cellule mémoire lue
	F7	Bit-resets sur la valeur de la cellule mémoire lue

TABLE 3.1 – Modèles de fautes considérés.

au chapitre 4 que `CELTIC` prend en compte l'injection de fautes multiples. Cependant, `CELTIC` n'implémente pas d'heuristique particulière pour accélérer l'injection de fautes multiples, il n'y a donc pas de différences techniques majeures entre la simulation d'une et plusieurs fautes.

3.2.4 Modèles de faute

`CELTIC` utilise des modèles au niveau ISA pour simuler les injections de faute. Ces modèles sont listés au tableau 3.1. On dénombre trois catégories principales, la corruption d'instruction, la corruption de registre et la corruption de mémoire, dans lesquelles 8 familles de modèles de faute :

- **Modèles de corruption du cache d'instruction** Ces modèles, inspirés des travaux de Rivière et al. [RNR+15], simulent une mauvaise mise à jour du cache d'instruction en rejouant les k dernières instructions exécutées à la place d'exécuter les k instructions suivantes.
- **Modèles de saut d'instruction** Ces modèles simulent des sauts d'une ou plusieurs instructions, comme observé expérimentalement par Dutertre et al. [DRPR19].
- **Modèles de bit-sets (bit-resets) sur l'instruction décodée** Ces modèles simulent un ou plusieurs bit-sets (bit-resets) dans l'instruction décodée, similairement aux effets observés par Colombier et al. [CMD+18].
- **Modèles de bit-sets (bit-resets) sur la valeur du registre lue** Ces modèles simulent un ou plusieurs bit-sets (bit-resets) sur la valeur du registre, similairement aux effets observés par Korak et al. [KH14].
- **Modèles de bit-sets (bit-resets) sur la valeur de la cellule mémoire lue** Ces modèles simulent un ou plusieurs bit-sets (bit-resets) sur la valeur de la cellule mémoire lue, comme observé expérimentalement par Dureuil et al. [DPdC+15].

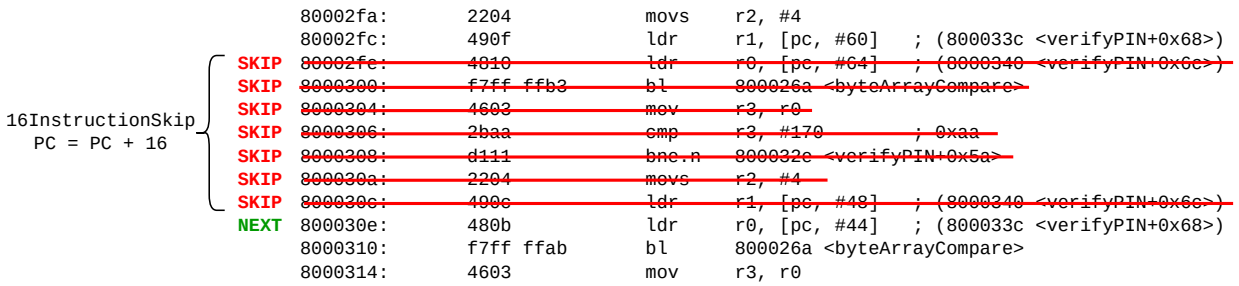


FIGURE 3.6 – Le modèle `16InstructionSkip` provoque le saut de six instructions 16-bit et d'une instruction 32-bit.

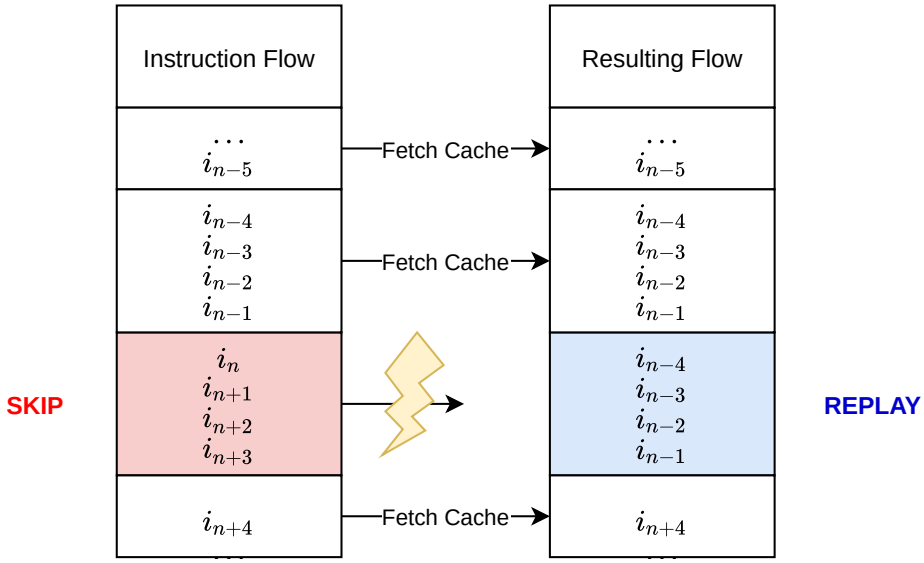
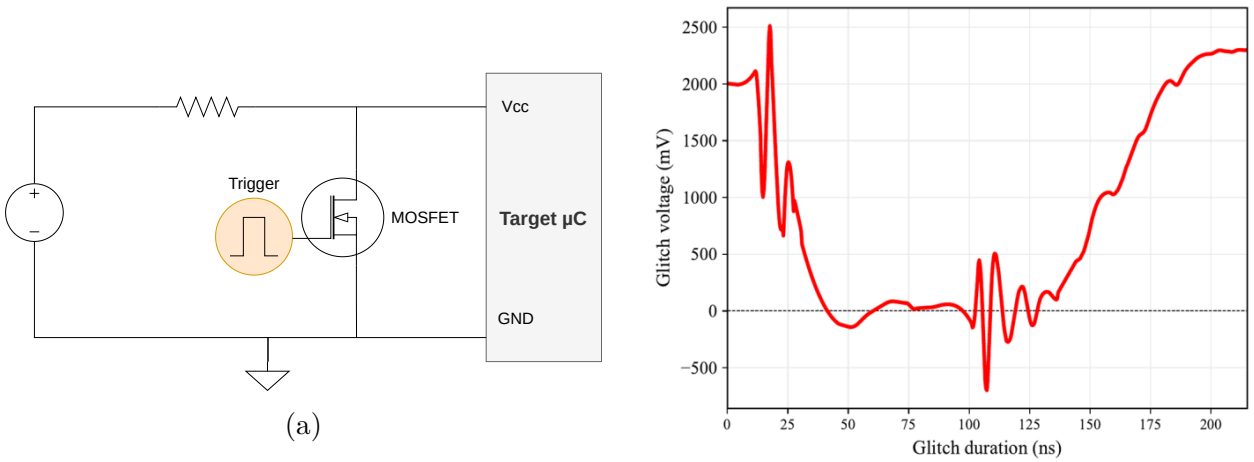


FIGURE 3.7 – Le modèle `4CacheCorruption` provoque le rejeu des quatre dernières instructions exécutées à la place des quatre suivantes.

En particulier, nous détaillons les modèles de sauts d'instruction et de corruption de cache qui seront utilisés au chapitre 4 :

- Les modèles de saut d'instruction, notés `kInstructionSkip`, simulent un saut de k octets lors du chargement de la prochaine instruction. Par exemple, le modèle `16InstructionSkip` est illustré à la figure 3.6.
- Les modèles de corruption de cache, notés `kCacheCorruption`, simulent le remplacement des k prochaines instructions par les k instructions exécutées précédemment. Par exemple, `4CacheCorruption` est illustré à la figure 3.7.

Pour terminer cette présentation de `CELTIC`, un exemple de script `Python` avec le modèle `16InstructionSkip` sur le `VerifyPIN_5` du benchmark FISSC [DPP+16] est détaillé à la figure A.I en annexe.



(a) (b)

FIGURE 3.8 – (a) Utilisation d'un transitor N-MOSFET pour générer des *power glitches* sur le µC en court-circuitant brièvement l'alimentation avec la masse, et (b) le *glitch* obtenu [BFP19].

3.3 Glitch Station

L'injection de faute par perturbation de la tension d'alimentation fait partie des premières techniques utilisées pour compromettre la sécurité d'un système [AK97]. Même si cette technique est relativement simple, en comparaison avec les perturbations par lumière focalisée, Timmers et al. [TSW16] ont montré qu'il était possible de générer des effets intéressants du point de vue d'un attaquant, comme notamment contrôler la valeur du compteur ordinal. Cependant, des limitations intrinsèques à cette technique nous ont motivé à concevoir la `GLITCH STATION` pendant cette thèse.

3.3.1 Motivation

Il existe de nombreuses techniques pour générer des *power glitches*. Une technique fréquemment utilisée consiste à utiliser un transistor MOSFET pour générer pendant un court instant un court-circuit entre l'alimentation et la masse (figure 3.8). Cette approche est utilisée dans des équipements d'injection de faute comme *ChipWhisperer* [OC14].

Pour rappel, au chapitre 2, nous avons déjà présenté les *power glitches*, et en particulier l'espace des paramètres associé à cette technique à la figure 2.2. Les deux paramètres principaux sont l'amplitude V_{glitch} et la durée T_{glitch} de la perturbation. La principale limitation de la génération de *glitches* avec un montage MOSFET est le manque de contrôle sur ces paramètres. Il est difficile de prédire et de contrôler la forme de *glitch* générée, car cette dernière dépend du MOSFET utilisé et du microcontrôleur ciblé. Pour résumer, le montage MOSFET permet de générer des perturbations de la tension d'alimentation, à bas coût, mais au détriment du contrôle de l'amplitude et de la durée de cette dernière.

D'autres outils, comme `GIAnT` [Osw16] ou `VC GLITCHER` [Ris17], se basent sur un FPGA (*Field-Programmable Gate Array*), et plus précisément le DAC intégré au FPGA, pour per-

Outils Existants	Contrôle	Prix	Technique
ChipWhisperer [OC14]	Faible	250 €	MOSFET
GiANT [Osw16]	Moyen	500 €	FPGA DAC
VC GLITCHER [Ris17]	Moyen	> 1000 €	FPGA DAC
Shapping the Glitch [BFP19]	Élevé	Prix du AWG + 100 €	AWG
Glitch Station	Élevé	100 €	R2-R DAC

TABLE 3.2 – Comparaison des outils existants en fonction du prix et du contrôle du *glitch*.

turber la tension d'alimentation. Un convertisseur numérique-analogique, appelé plus communément DAC, est un composant électronique permettant de transformer une valeur numérique (codée sur plusieurs bits) en une valeur analogique (une tension). Ce type de dispositif est notamment utilisé dans les cartes son. Dans notre contexte, le DAC permet de convertir la forme de la perturbation encodée numériquement vers une chute brève et brutale de la tension. Contrairement au montage MOSFET, le DAC permet de contrôler finement l'amplitude et la durée de la perturbation. Notez que si `GIAnT` est un outil communautaire, `VC GLITCHER` est un produit commercialisé par Riscure.

Une récente approche consiste à utiliser un générateur de signaux arbitraires (*Arbitrary Waveform Generator*, AWG) pour générer le *glitch*. Un AWG étant en partie composé d'un DAC, cette technique peut sembler très similaire à celle précédente. Seulement, au lieu de définir la forme du *glitch* uniquement en fonction de l'amplitude et de la durée, Bozzato et al. [BFP19] définissent la forme du de la perturbation avec 8 points distincts (similairement à figure 3.10), permettant de contrôler très précisément les formes générées. Le prix de ce montage dépend fortement de l'AWG choisi.

Enfin, la `GLITCH STATION` est un outil développé pendant cette thèse s'inspirant de l'approche de Bozzato et al. Mais au lieu d'utiliser un AWG, nous avons choisi d'utiliser une carte de développement `NUCLEO-H743ZI2` pour générer la forme du glitch, à l'aide d'un montage R2-R qui sera détaillé à la section suivante, permettant ainsi de réduire les coûts. Le tableau 3.2 résume les outils existants en fonction du prix et du contrôle du *glitch*.

3.3.2 Architecture

L'idée derrière ce projet était de concevoir un outil facile d'utilisation, à bas coût, mais offrant suffisamment de contrôle sur la forme du *glitch*. Pour ce faire, la `GLITCH STATION` est articulée autour de deux éléments, la `GLITCH MASTER` et la `GLITCH UNIT`, comme presenté à la figure 3.9.

3.3.2.1 Glitch Master

La `GLITCH MASTER` est chargée de calculer la forme de la perturbation, puis de l'envoyer à la `GLITCH UNIT` qui sera chargée de la générer. Concrètement, la `GLITCH MASTER` désigne le PC de bureau exécutant un script `Python` chargé de s'interfacer avec la `GLITCH UNIT` et

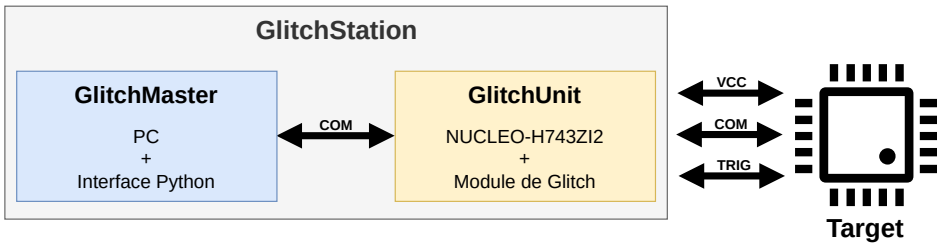


FIGURE 3.9 – Vue d'ensemble de la `GLITCH STATION`.

		Parameter	
		$x_0 ... x_7$	*duration*
Digital	Range	$[\![80, 180]\!]$	$[\![8, 128]\!]$
	Resolution	8-bit	8-bit
Analog	Range	$[0.6, 5.1]$	$[0.2, 3.2]$
	Unit	V	µs

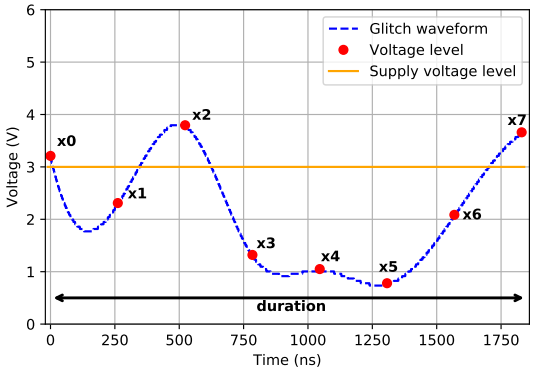


FIGURE 3.10 – Espace de paramètre de la `GLITCH STATION`, $\approx 10^{18}$ configurations. La forme du *glitch* est définie avec 8 niveaux de tension ($x_0 ... x_7$) et la durée.

de contrôler la forme du prochain *glitch* à injecter sur la cible.

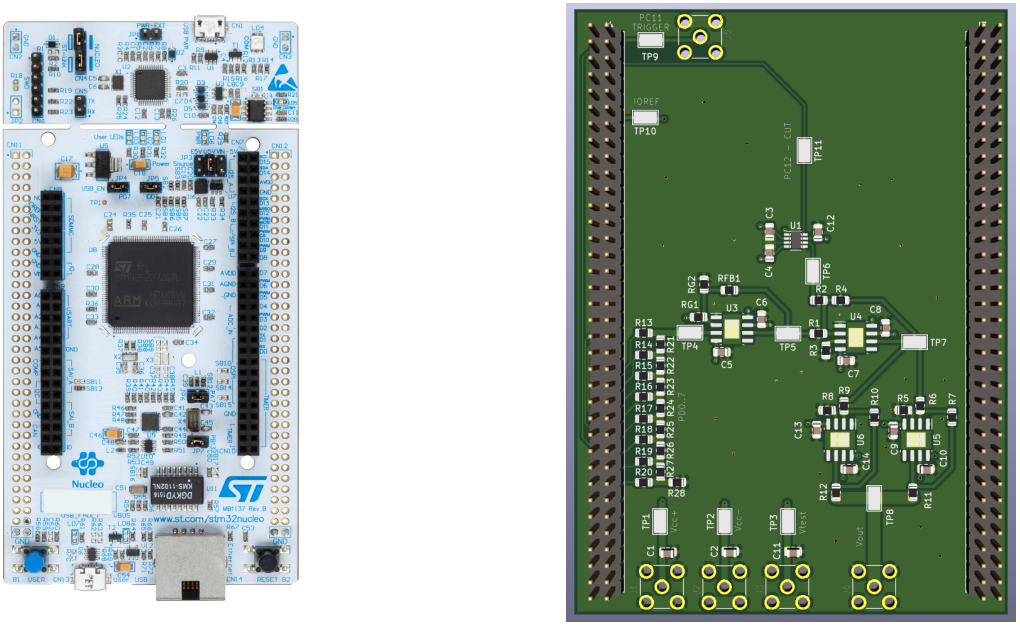
Comme illustré sur la figure 3.10, la forme de la perturbation peut être ajustée avec 8 niveaux de tensions ($x_0 ... x_7$) par défaut, en plus de la durée de la perturbation. La forme calculée est une interpolation entre les points $x_0 ... x_7$. Si les points $x_0 ... x_7$ permettent de contrôler précisément la forme du *glitch* pour s'adapter à une grande variété de microcontrôleurs, la taille de l'espace des paramètres explose, avec $\approx 10^{18}$ configurations possibles.

Ainsi, pour identifier les paramètres d'équipement induisant des fautes sur le microcontrôleur ciblé, des techniques d'optimisation plus efficaces que la recherche aléatoire ou la recherche par quadrillage doivent être utilisées. À ce jour, la `GLITCH STATION` implémente 4 techniques d'optimisation différentes : la recherche aléatoire (voir chapitre 2), l'optimisation par algorithme génétique (voir chapitre 2), l'optimisation par algorithme de bandit (voir chapitre 6) et l'optimisation bayésienne (voir chapitre 6). Un exemple de script `Python` est présenté à la figure A.II en annexe.

3.3.2.2 Glitch Unit

Une fois la forme de la perturbation calculée et envoyée par la `GLITCH MASTER`, la `GLITCH UNIT` s'occupe de la génération physique du *glitch*. Pour ce faire, la `GLITCH UNIT` s'articule autour d'une `NUCLEO-H743ZI2` surmontée d'une carte électronique chargée de la génération et de l'amplification de la perturbation, comme illustré à la figure 3.11.

Le fonctionnement de l'application chargée dans la `NUCLEO-H743ZI2` est relativement



(a) (b)

FIGURE 3.11 – (a) La NUCLEO-H743ZI2 et (b) la carte électronique chargé de la génération et de l'amplification du *glitch*.

simple et ne sera pas détaillé. Concrètement, la carte reçoit des commandes de la `GLITCH MASTER`, les exécute puis retourne les résultats à la `GLITCH MASTER`.

Le schéma de la carte électronique est présenté à la figure 3.12. On peut distinguer trois blocs principaux :

- **Le DAC par montage R2-R** En utilisant un réseau de résistances, on peut facilement et de manière économique convertir un signal numérique en un signal analogique [Ken00]. Le signal numérique provient de 8 ports entrées-sorties de la NUCLEO-H743ZI2, par transfert DMA (*Direct Memory Access*). Le signal analogique obtenu est compris entre 0 et 3.3V. Ce montage permet d'atteindre $\approx$ 30Msps, c'est-à-dire que la `GLITCH STATION` peut générer des *glitches* d'une durée au plus courte de $\approx$ 30 ns.

- **Le convertisseur unipolaire vers bipolaire** Le convertisseur unipolaire vers bipolaire permet de transformer le signal de sortie du DAC, compris entre 0 et 3.3V, vers un signal compris entre -6V et 6V. Le montage se base sur un amplificateur opérationnel THS3491 (slew rate 8000 V/µs, bandwidth 900 MHz).

- **Le montage additionneur et suiveur** Enfin la tension d'alimentation du microcontrôleur ciblé V_{cc} est additionnée au signal en sortie du convertisseur unipolaire vers bipolaire. Le signal final est donc compris entre -6V + V_{cc} et 6V + V_{cc}. La `GLITCH UNIT` permet donc à la fois d'alimenter en continu la cible et de générer le *glitch* au moment opportun.

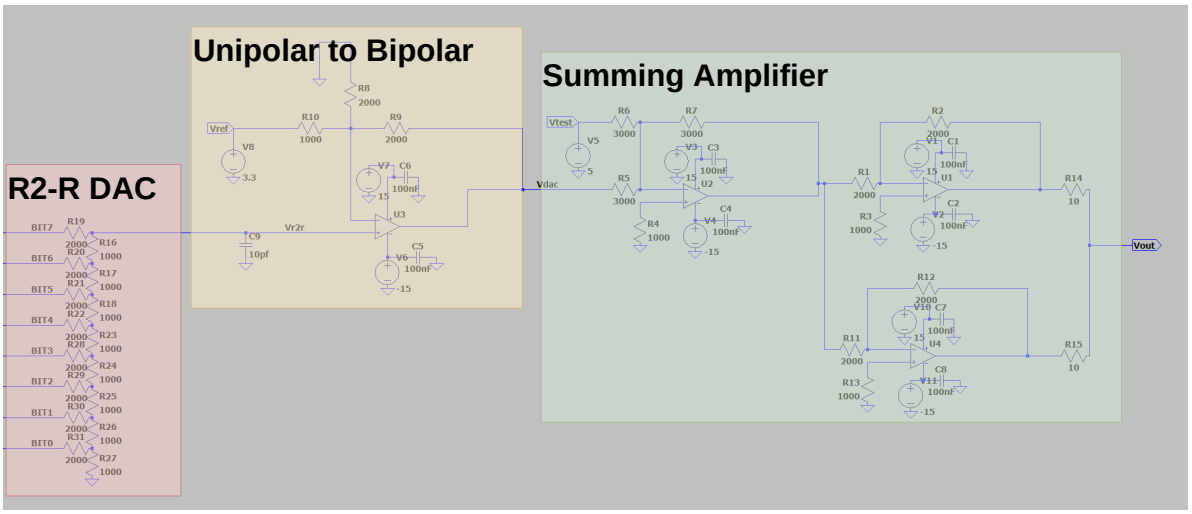


FIGURE 3.12 – Schéma de la carte électronique chargée de la génération du *gltich*. On distingue les trois blocs principaux, le DAC R2-R, la conversion du signal unipolaire en signal bipolaire et le montage additionneur.

Pour résumer, la `GLITCHSTATION` permet de contrôler finement la forme de la perturbation avec 8 niveaux de tensions distincts, contrairement à `VC GLITCHER` et `GiANT` qui ne contrôle que l'amplitude maximale du *glitch*. Ensuite, la `GLITCHSTATION` est compacte, facile d'utilisation et ne nécessite pas d'équipement externe, comme un AWG, contrairement à l'outillage proposé par Bozzato et al [BFP19]. Cela permet de baisser les coûts et d'avoir un outil facilement et rapidement déployable pour une évaluation de sécurité.

3.3.2.3 Améliorations

Une amélioration possible serait de remplacer le montage R2-R DAC 8-bit par un DAC AD9102 14-bit de chez Analog à 15€ pour atteindre les 180 Msps, soit un glitch d'une durée au plus courte de $\approx$ 5 ns, soit une durée 6 fois plus courte qu'avec le montage R2-R actuel. Néanmoins, cela n'est peut-être pas nécessaire à court terme, car nous n'avons pas rencontré de difficulté à générer des fautes sur une grande variété de microcontrôleurs avec le montage R2-R. D'autant plus que le montage du DAC AD9102 est plus complexe, nécessitant une horloge de référence externe.

3.4 Conclusion

Dans ce chapitre, nous avons présenté les outils qui ont été utilisés pendant cette thèse, à savoir `CELTIC` et la `GLITCH STATION`. D'une part, `CELTIC` permet de simuler des injections de fautes au niveau ISA pour identifier de potentielles vulnérabilités et d'autre part, la `GLITCH STATION` nous permet d'injecter des fautes facilement et à moindre coût. En particulier, ces deux outils seront utilisés dans le chapitre suivant pour identifier et exploiter des vulnérabilités à l'injection de fautes multiples.

Chapitre 4

Méthodologie pour l'identification et l'exploitation de vulnérabilités à l'injection de fautes multiples sur microcontrôleur

Sommaire

4.1 Introduction

Dans le chapitre 2, nous avons présenté, entre autres, plusieurs méthodologies pour combler l'écart entre différents niveaux d'évaluations. Pour aller plus loin, nous proposons une méthodologie pour l'identification et l'exploitation de vulnérabilité à l'injection de fautes multiples.

En particulier, notre méthode est conçue pour être utilisée de bout-en-bout, de la caractérisation de la cible, à l'exploitation d'injections de faute simple et multiple. Elle se divise en trois étapes, à savoir, 1) l'inférence de modèles de faute, 2) l'identification des vulnérabilités et enfin 3) l'exploitation des vulnérabilités. L'avantage principal de notre méthode est d'automatiser chacune de ces étapes pour 1) accélérer l'évaluation de sécurité, 2) augmenter la répétabilité des résultats et 3) systématiser le processus d'évaluation.

Au niveau expérimental, notre méthodologie est adaptable à la majorité des situations rencontrées lors d'évaluations de sécurité. Plusieurs techniques d'injections de fautes ont été évaluées sur 3 références de microcontrôleurs différentes. Par ailleurs, notre méthodologie a pu identifier une faiblesse d'implémentation dans une des macros de l'outil CFI-C [HLB19], permettant de générer automatiquement des contre-mesures d'intégrité de flot de contrôle au niveau source. Enfin, notre méthodologie permet d'exploiter une vulnérabilité à l'injection double faute jusqu'à 30 fois plus rapidement qu'une approche uniquement basée sur une caractérisation.

Après avoir préciser les motivations et les principaux défis pour identifier et exploiter des fautes multiples à la section 4.2, nous présenterons notre approche en détail à la section 4.3. La section 4.4 résume les expériences conduites avec notre méthodologie, et précise notamment les microcontrôleurs et les applications cibles retenues. En particulier, à la section 4.5, nous détaillons deux de ces expériences, illustrant la versatilité de notre méthodologie. Ensuite, à la section 4.6, nous évaluons notre méthodologie sur plusieurs critères, la couverture de fautes, la détection de vulnérabilités, et la vitesse d'exploitation. Enfin, à la section 4.7, les possibles pistes d'amélioration sont abordées.

Les travaux de ce chapitre ont été publiés à la conférence FDTC en septembre 2020 [WMP20] et sont en cours de soumission dans la revue *The Journal of Cryptographic Engineering*.

4.2 Motivations

Avant de présenter notre méthodologie, nous détaillons les principaux défis liés à l'injection de faute multiples, puis nous revenons sur les méthodologies présentées au chapitre 2 et leur positionnement face à ces problématiques.

4.2.1 Défis

Avec l'évolution des attaques par faute au cours de la dernière décennie, et en particulier des attaques par fautes multiples, une évaluation de sécurité doit maintenant prendre en compte les scénarios d'attaque basés sur plusieurs injections de faute [Joi20].

Néanmoins, si identifier et exploiter des vulnérabilités à l'injection d'une seule faute est déjà une première épreuve, le fait d'en considérer plusieurs complique significativement la tâche. La raison principale est l'explosion combinatoire du nombre de paramètres d'attaque directement lié avec le nombre de fautes injectées. Par exemple, avec la `GLITCH STATION`, la taille de l'espace de paramètres d'équipement est de 10^{18} (chapitre 3). En comptant un balayage temporel de 100 délais d'injection pour couvrir une application cible, cela nous

donne 10^{20} paramètres d'attaque différents en simple faute. En double faute, on passe à 10^{40} paramètres d'attaque.

Comme il n'est pas possible de tester tous les paramètres d'attaque pendant une évaluation de sécurité en temps contraint, y compris en faute simple, l'évaluateur va, d'une part 1) *identifier des vulnérabilités* pour cibler les zones critiques de l'application de manière à réduire l'étendue du balayage temporel, et d'autre part 2) *caractériser le microcontrôleur ciblé* (chapitre 2) afin de réduire l'espace des paramètres d'équipement.

Si cela permet de réduire efficacement l'espace des paramètres d'attaque, l'identification de vulnérabilités et la caractérisation introduisent de nouveaux défis, en particulier pour les fautes multiples :

Défi D1 : Identifier des modèles de faute spécifiques Pour identifier de potentielles vulnérabilités sur l'application cible, il est courant d'utiliser des modèles de faute génériques (chapitre 2) pour construire des chemins d'attaque. Malheureusement, l'utilisation de modèles génériques conduit généralement aux situations suivantes :

- Les modèles génériques ne tiennent pas compte des spécificités du microcontrôleur cible, et ainsi certains chemins d'attaque identifiés ne seront pas réalisables en pratique. Cela se traduit par une perte de temps tout d'abord pendant l'identification des vulnérabilités, mais aussi pendant l'exploitation des vulnérabilités avec l'équipement d'injection de faute.
- À l'inverse, comme les spécificités du microcontrôleur cible ne sont pas prises en compte par les modèles génériques, certaines vulnérabilités ne seront pas détectées. Avec la découverte de nouveaux modèles de faute de plus en plus complexes [DRPR19, RNR+15, CMD+18, LBD+19], il est donc capital de ne pas se contenter uniquement des modèles génériques.
- Dans le cas de fautes multiples, l'explosion combinatoire ne fait qu'empirer les deux points précédents : plus de chemins d'attaque irréalisables expérimentalement et plus de vulnérabilités non détectées. Dans ce contexte, il est indispensable de tenir compte des spécificités du microcontrôleur.

Si l'identification de modèles spécifiques au microcontrôleur ciblé est une priorité pour réussir des attaques complexes, il faut au préalable réussir à diagnostiquer et modéliser l'effet des fautes sur la cible. Cependant, cela est le plus souvent fait manuellement, ce qui peut conduire à des erreurs ou des oublis, en plus d'être chronophage.

Défi D2 : Utiliser des modèles de faute différents en multi-fautes Pour trouver de nouveaux chemins d'attaque, on peut utiliser un modèle de faute différent à chacune des injections. En particulier, il est théoriquement possible de construire des attaques complexes avec des sauts d'instruction de différentes longueurs [PCHR20]. Mais cela a un coût non négligeable, à savoir augmenter considérablement le nombre de chemins d'attaque à analyser. Comme précédemment, cela augmente à la fois le temps passé pendant l'identification et

l'exploitation des vulnérabilités. Enfin, il faut aussi s'assurer d'utiliser des modèles spécifiques à la cible, car dans le cas contraire, une grande majorité des chemins d'attaque identifiés ne seront pas réalisables en pratique.

Défi D3 : Sélectionner les paramètres d'attaque optimaux Si l'utilisation de modèles spécifiques à la cible permet de réduire l'écart entre l'identification et l'exploitation des vulnérabilités, il reste à sélectionner les paramètres d'attaque, à savoir les paramètres d'équipement et les délais d'injection. Pour obtenir des résultats reproductibles rapidement, les paramètres d'équipement maximisant la probabilité d'exploiter les vulnérabilités identifiées doivent être sélectionnés en priorité. Cela nécessite, entre autres, de réussir à lier les paramètres d'équipement aux modèles de fautes.

4.2.2 Méthodologies existantes

Les méthodologies existantes, présentées au chapitre 2, proposent des stratégies différentes pour réduire l'écart entre un ou plusieurs niveaux d'analyse. Dans ce qui suit, nous analysons leurs positionnements par rapport aux défis énoncés au paragraphe précédent.

Tout d'abord, Rivière et al. [RPL+14] combinent une analyse au niveau code source et matériel pour réduire la zone temporelle à cibler. Les fautes injectées sont injectées artificiellement sur la cible avec l'outil EFS (*Embedded Fault Simulator*) ce qui ne répond que partiellement au *Défi D3*. De plus, les modèles spécifiques au microcontrôleur ciblé ne sont pas identifiés (*Défi D1*). Bien qu'il soit possible de faire du multi-fautes, il n'est pas possible de varier les modèles utilisés en multi-fautes (*Défi D2*) en utilisant leur stratégie.

Dureuil et al. [DPdC+15] proposent une approche combinant une analyse au niveau binaire et matériel. L'accent est mis sur l'inférence de modèles de faute probabilistes, spécifiques au microcontrôleur ciblé (*Défi D1*). Ces modèles sont ensuite utilisés pour évaluer la robustesse de la cible en simulation. Si potentiellement il serait possible de sélectionner des paramètres d'attaque à l'aide des modèles probabilistes, cet aspect n'est pas détaillé dans leur approche (*Défi D3*). Enfin, les fautes multiples n'étant considérées, il n'est pas possible de construire des attaques complexes avec plusieurs modèles de faute différents (*Défi D2*).

Given et al. [GWJL18] proposent d'identifier les modèles spécifiques à la cible en combinant une analyse au niveau binaire et matériel (*Défi D1*). Des modèles complexes basés sur une combinaison de plusieurs modèles génériques sont utilisés à ces fins. Cependant, comme les fautes multiples ne sont pas envisagées, il n'est pas possible de répondre au *Défi D2*. Également, s'il n'y a pas de sélection des paramètres d'attaque tel que défini dans le *Défi D3*, leur stratégie permet tout de même de cibler temporellement les zones d'intérêts pour l'analyse au niveau matériel.

Laurent et al. [LDPPB21] se basent sur des simulations d'injection de faute au niveau RTL pour mettre en évidence des modèles complexes de fautes spécifiques à l'architecture ciblée (*Défi D1*). L'analyse se poursuit alors au niveau binaire en utilisant les modèles identifiés. Les fautes multiples n'étant pas considérées, il n'est pas possible de répondre au *Défi D2*. Enfin, comme il n'y a pas d'analyse au niveau matériel, il n'est pas possible de sélectionner les paramètres d'attaque (*Défi D3*).

	Niveau d'analyse	Fautes Multiples	D1	D2	D3
Rivière et al. [RPL+14]	Code Source Matériel	✓	✗	✗	✗
Dureuil et al. [DPdC+15]	Binaire Matériel	✗	✓	✗	✗
Given et al. [GWJL18]	Code Source Matériel	✗	✓	✗	✗
Laurent et al. [LDPPB21]	Code Source RTL	✗	✓	✗	✗
Alshaer et al. [ACD+21]	Code Source RTL Matériel	✗	✓	✗	✗

TABLE 4.1 – Comparaison des méthodologies existantes selon les défis D1, D2 et D3.

Alshaer et al. [ACD+21] étend la méthodologie de [LDPPB21] en rajoutant une analyse au niveau matériel. On retrouve donc l'identification de modèles spécifiques à la cible (*Défi D1*). Pour le moment, la méthode a été appliquée seulement avec des injections de faute simple (*Défi D2*) et l'aspect de sélection des paramètres d'attaque (*Défi D3*) n'est pas détaillé.

Le tableau 4.1 compare les méthodologies détaillées au chapitre 2 en fonction des défis D1, D2 et D3. La méthodologie que nous proposons vise à répondre aux trois défis énoncés en mettant l'accent sur l'identification et l'exploitation des vulnérabilités aux fautes multiples.

4.3 Notre approche

4.3.1 Vue générale

La méthodologie proposée repose en grande partie sur l'identification des modèles spécifiques les plus probables pour le microcontrôleur ciblé (*Défi D1*). D'une part, diminuer considérablement le nombre de modèles de fautes à prendre en compte permet de réduire la complexité de la simulation d'injection de faute. D'autre part, se baser uniquement sur les modèles les plus probables permet de faciliter l'exploitation des vulnérabilités identifiées lors de la simulation.

L'autre point central de notre approche est d'évaluer séparément dans un premier temps le microcontrôleur et l'application, dans le but d'identifier les paramètres d'attaque optimaux (*Défi D3*). Ensuite, dans un deuxième temps, la cible complète (microcontrôleur et application) est évaluée avec les paramètres d'attaque sélectionnés.

Enfin le dernier aspect important de notre méthodologie est de prendre en considération l'injection de fautes multiples, et en particulier identifier des chemins d'attaque exotiques utilisant des modèles différents pour chaque injection de faute (*Défi D2*).

Pour résumer, la figure 4.1 détaille notre approche qui se décompose en trois étapes clés :

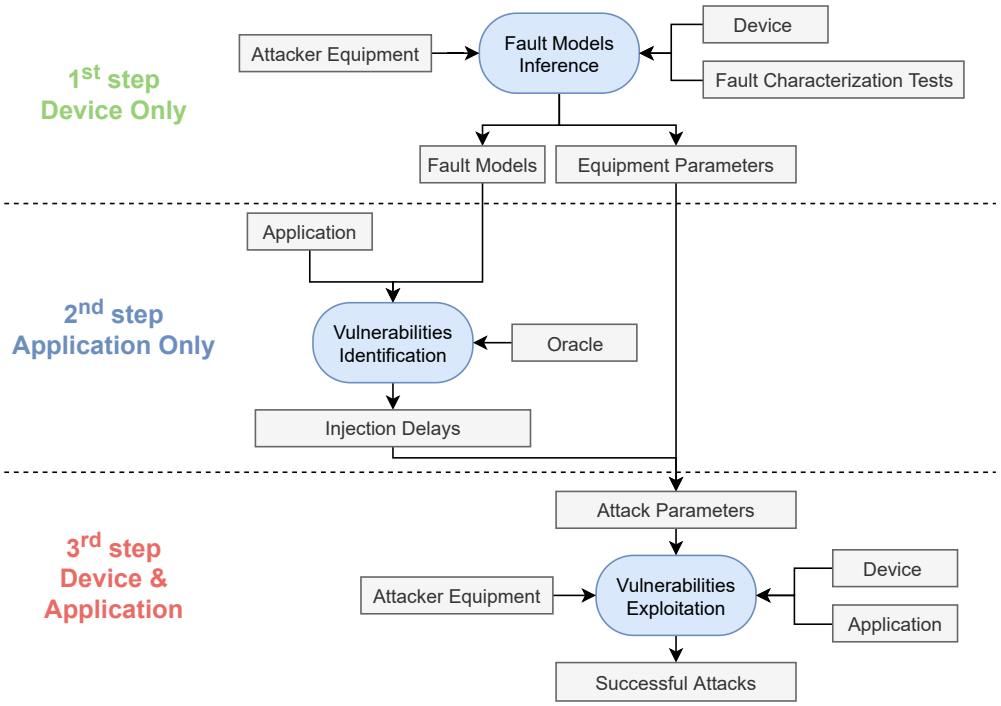


FIGURE 4.1 – Vue globale de notre approche.

Première étape : Inférence de modèles de faute D'une part, des *modèles de faute* spécifiques au microcontrôleur ciblé sont inférés automatiquement en croisant les résultats fautés obtenus à partir d'une caractérisation et d'une simulation d'injection de faute. D'autre part, les *paramètres d'équipement* les plus probables, selon les modèles de faute inférés, sont identifiés.

Deuxième étape : Identification des vulnérabilités En considérant uniquement les modèles spécifiques les plus probables, l'application est évaluée automatiquement avec `CELTIC` dans le but d'identifier les *délais d'injection* pouvant potentiellement conduire à de potentielles failles de sécurité sur la cible.

Troisième étape : Exploitation des vulnérabilités Les *paramètres d'attaque* sont générés automatiquement à partir des informations précédentes, à savoir les paramètres d'équipement et les délais d'injection. Les paramètres d'attaques sont sélectionnés proportionnellement à leur probabilité de faute jusqu'à ce qu'une attaque réussisse.

Si notre méthodologie proposée est inspirée en partie des travaux de Dureuil et al. [DPdC+15], plusieurs différences clés sont à noter. Tout d'abord, contrairement à leur approche, notre inférence de modèle de faute est automatisée, ce qui est plus robuste et beaucoup moins contraignant. Ensuite, dans leur approche, les vulnérabilités identifiées avec `CELTIC` ne sont pas exploitées *en pratique*. Et enfin, la problématique des fautes multiples n'y est pas abordée.

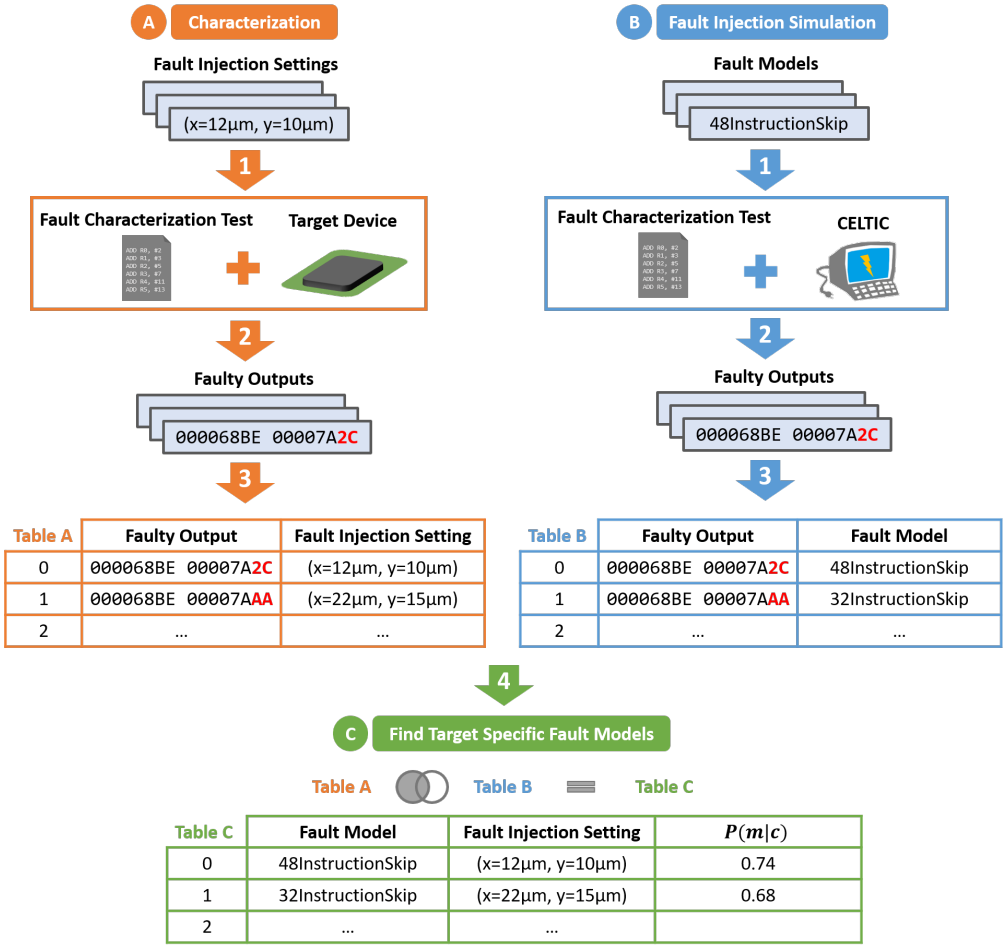


FIGURE 4.2 – Vue globale de la première étape d'inférence de modèles de faute. Cette étape se divise en trois sous-étapes, (A) la caractérisation de fautes, (B) la simulation d'injection de faute et (C) l'identification des modèles spécifiques.

Dans la suite de cette section, nous détaillons chaque étape, puis notre méthodologie sera illustrée sur des exemples à la section 4.5.

4.3.2 Première étape : Inférence de modèles de faute

L'objectif de la première étape est d'inférer des modèles de faute propres au microcontrôleur ciblé. Ces modèles de fautes identifiés seront utilisés par la suite pour identifier des vulnérabilités à l'injection de fautes sur l'application cible. Cette étape consiste à croiser les résultats expérimentaux de la caractérisation de fautes du microcontrôleur ciblé avec les résultats simulés de `CELTIC`. Cette étape nécessite des tests de caractérisation de fautes, qui seront utilisés à la fois pour la caractérisation et la simulation d'injection de fautes avec `CELTIC`. L'application cible n'est donc pas utilisée lors de la première étape.

4.3.2.1 Caractérisation de fautes

La caractérisation de fautes permet, d'une part, d'identifier les configurations d'équipement permettant de maximiser le nombre de résultats fautés, et d'autre part, de comprendre le comportement du microcontrôleur en réponse aux injections de fautes. Comme l'application cible ne permet généralement pas d'identifier facilement et distinctement les effets des fautes, des tests de caractérisation de fautes sont utilisés à la place pour propager les effets des fautes sur le microcontrôleur. Ces tests, déjà présentés dans le chapitre 2, sont le plus souvent constitués d'une suite d'instructions arithmétique et logique [RNR+15, PHM+19, CMD+18, DRPR19] ou d'instructions *load* et *store* [TSW16, DPdC+15]. L'influence des tests sur la propagation des fautes sera abordée au chapitre 5.

La partie A de la figure 4.2 détaille le processus de la caractérisation de fautes. Tout d'abord, différents paramètres d'injection de faute sont testés (1) sur le microcontrôleur pendant l'exécution du test de caractérisation de fautes, en prenant soin de conserver les résultats fautés (2). Une table liant les résultats fautés avec les paramètres d'injection de faute sera ensuite générée (3). Notez que différentes techniques d'optimisation, détaillées dans le chapitre 2, peuvent être utilisées pour explorer efficacement l'espace des paramètres d'injection de faute. Par ailleurs, de nouvelles techniques d'optimisation seront détaillées au chapitre 6.

4.3.2.2 Simulation d'injection de faute

Une des difficultés récurrentes lors de la caractérisation de fautes est de diagnostiquer l'effet de la faute à l'origine du résultat fauté, et de généraliser à partir de ces observations des modèles de faute. Cette étape est généralement faite manuellement, ce qui est fastidieux et chronophage, en plus d'être facilement enclin aux erreurs et oublis. Pour résoudre ce problème, une table liant résultats fautés et modèles de faute est automatiquement générée, ce qui permettra, dans un second temps, de trouver rapidement le(s) modèle(s) de faute conduisant à un résultat fauté donné.

Plus précisément, notre simulateur d'injection de faute `CELTIC` est utilisé pour émuler l'architecture du microcontrôleur ciblé et simuler des injections de faute pendant l'exécution du test de caractérisation de fautes. La partie B de la figure 4.2 présente une vue générale du processus. Tout d'abord, `CELTIC` simule des injections de fautes selon un ensemble de modèles de faute (1), et sauvegarde chaque résultat fauté (2). Enfin, une table liant résultats fautés et modèles de faute est générée (3). Par ailleurs, la simulation d'injection de faute peut se faire en parallèle de la caractérisation de fautes pour économiser du temps. L'ensemble de modèles simulés par défaut est détaillé dans le tableau 3.1 du chapitre 3.

4.3.2.3 Combinaison des résultats

En combinant les résultats fautés expérimentaux avec les résultats fautés simulés, il est facile d'inférer les modèles de faute spécifiques au microcontrôleur ciblé. Plus précisément, on génère une table liant les modèles de faute spécifiques au microcontrôleur ciblé avec les configurations d'équipement. La table liant modèles et configurations (figure 4.2) est obtenue

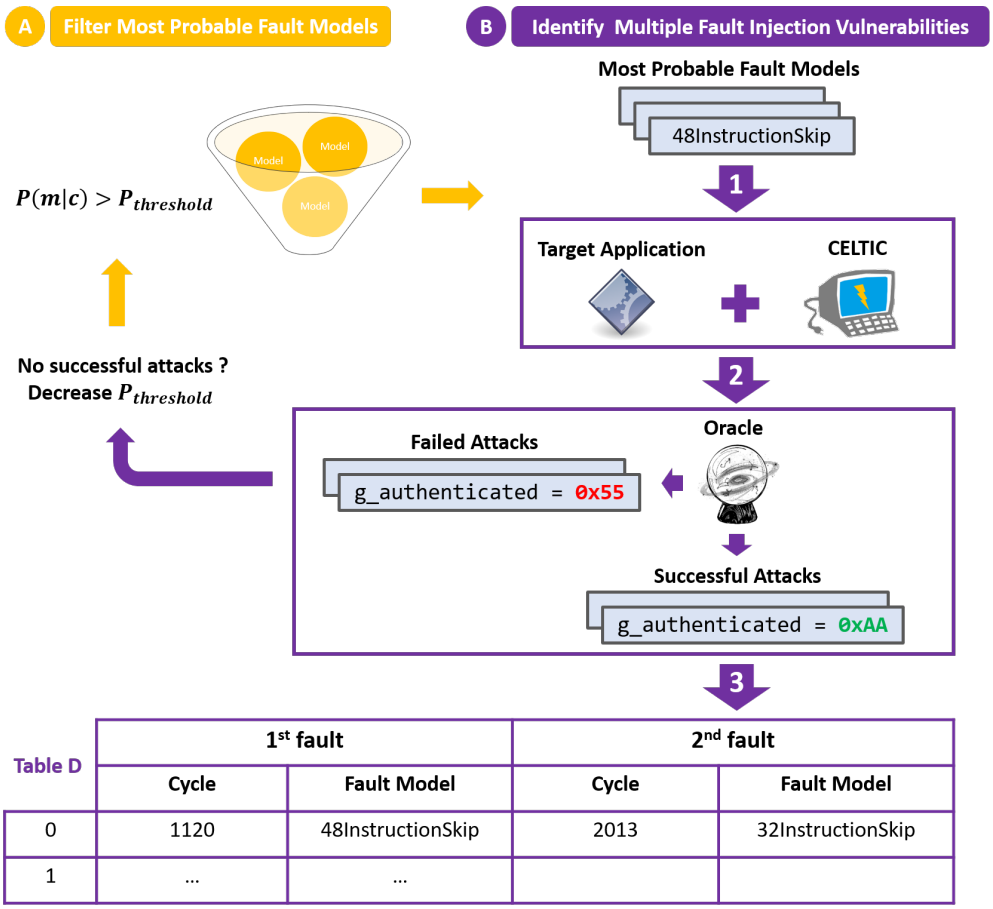


Table D

	1st fault		2nd fault	
	Cycle	Fault Model	Cycle	Fault Model
0	1120	48InstructionSkip	2013	32InstructionSkip
1	...	...		

FIGURE 4.3 – Vue globale de la deuxième étape d'identification des vulnérabilités par simulation. Cette étape se divise en deux sous-étapes, (A) les modèles les probables sont filtrés, et (B) les attaques par fautes multiples sont identifiées

par jointure à gauche sur les résultats fautés de la table de la caractérisation avec la table de la simulation.

De plus, on associe à chaque couple modèle-configuration une probabilité $P(m|c)$, similairement à Dureuil et al. [DPdC+15], désignant la probabilité d'obtenir l'effet de faute décrit par le modèle m avec la configuration c. Cela permettra ultérieurement de filtrer les meilleurs modèles/configurations.

4.3.3 Deuxième étape : Identification des vulnérabilités par simulation

Cette étape utilise seulement les modèles de faute les plus probables, inférés à l'étape précédente, pour simuler des injections de faute réalistes.

4.3.3.1 Filtrage des modèles les plus probables

La simulation d'injection de fautes multiples indépendantes entraine une explosion combinatoire des chemins d'attaque. Par conséquent, le nombre de modèles de faute simulés avec `CELTIC` est volontairement limité. Pour cela, les modèles de faute les plus probables sont filtrés en se basant sur les résultats expérimentaux précédents (figure 4.3).

L'évaluateur sélectionne les modèles avec une probabilité $P(m|c) > P_{threshold}$, $P_{threshold}$ étant un seuil choisi arbitrairement. Diminuer le seuil $P_{threshold}$ augmente le nombre de modèles sélectionnés, ce qui permet d'améliorer l'exhaustivité de l'analyse de vulnérabilité avec `CELTIC` mais conduit à rallonger le temps de simulation et diminue la vraisemblance des vulnérabilités identifiées. Ici, l'ajustement du seuil se fait de manière itérative : le seuil diminue progressivement pour augmenter le nombre de modèles sélectionnés jusqu'à obtenir une attaque réussie.

4.3.3.2 Simulation d'injection de fautes multiples indépendantes

L'analyse de vulnérabilité à l'injection de fautes multiples est automatisée avec `CELTIC`, qui permet de trouver exhaustivement les attaques réussies par rapport à l'oracle et en fonction des modèles sélectionnés (partie B de la figure 4.3). Si l'accent est mis sur les fautes multiples, les vulnérabilités à l'injection de faute unique sont également prises en compte.

Algorithme 1 : FindMFA pseudocode

Function `FindMFA`(X, n, H) **is**
- **Input :** An execution trace X, the fault injection history H, the set of fault models M, the attack order n and the oracle `Oracle`.
- **Output :** The successful attacks S.
- $S \leftarrow \emptyset$;
- **for** $i \leftarrow 1$ **to** $|X|$ **do**
  - **foreach** $m \in M$ **do**
    - $X' \leftarrow$ `SimulateFI`(x_i, m);
    - $H' \leftarrow H \cup \{i, m\}$;
    - **if** `Oracle`(X') **then**
      - $S \leftarrow S \cup \{H'\}$;
    - **else if** $n > 1$ **then**
      - $S \leftarrow S \cup$ `FindMFA`$(X', n-1, H')$;
- **return** S;

`CELTIC` simule des injections de fautes à partir des modèles sélectionnés (1). Chaque trace d'exécution fautée est analysée par l'oracle (2). Lorsqu'une attaque réussie est identifiée, le cycle d'exécution de l'instruction fautée et le modèle de faute associé sont sauvegardés dans une table (3).

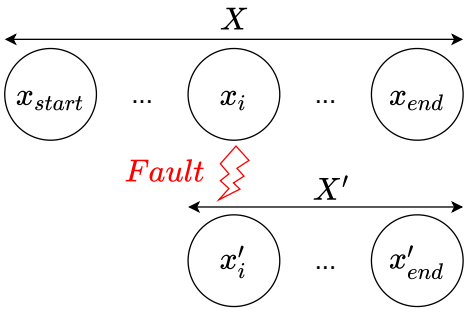


FIGURE 4.4 – Génération d'une trace d'exécution fautée X' avec `CELTIC` à partir d'une trace de référence X.

L'Algorithme 1 détaille le pseudocode de `CELTIC` pour identifier des vulnérabilités à l'injection de fautes multiples. Tout d'abord, `CELTIC` génère une trace d'exécution de l'application de référence, c'est-à-dire sans faute.

Ensuite, la fonction `FindMFA` est appelée. Cette fonction prend comme arguments une trace d'exécution de l'application X, un ensemble de modèles M, un historique des injections de fautes précédentes H, le nombre de fautes n (e.g. $n = 2$ pour l'injection de double faute), et enfin l'oracle `Oracle` définit par l'évaluateur. Lors du premier appel de `FindMFA`, X est l'exécution de référence et l'historique H est vide.

Pour chaque état x_i de la trace d'exécution X et pour chaque modèle m, `CELTIC` simule une injection de faute selon le modèle m à l'état x_i et retourne une nouvelle trace d'exécution fautée X' démarrant à l'état fauté x'_i (figure 4.4). L'historique H est mis à jour dans une variable locale H'. Concrètement, le cycle d'exécution de l'instruction i ainsi que le modèle de faute m sont sauvegardés.

Si la trace X' est identifiée comme étant une attaque réussie par l'oracle, les informations concernant l'attaque (cycles et modèles de faute de l'historique H') sont enregistrées dans une table S (figure 4.3). Sinon, n est décrémenté et le processus continu avec la trace X' jusqu'à ce que toutes les fautes soient injectées. À la fin, la table S contenant les cycles et modèles de faute des attaques réussies est retournée par `FindMFA`.

4.3.4 Troisième étape : Exploitation des vulnérabilités

L'écart entre la simulation et la réalité est souvent difficile à combler. Cette dernière étape permet d'assister l'évaluateur à exploiter les vulnérabilités précédemment identifiées avec `CELTIC` plus facilement. L'équipement est configuré avec les meilleurs réglages identifiés lors de la caractérisation et le balayage temporel se concentre sur les zones critiques identifiées avec `CELTIC`.

4.3.4.1 Génération des paramètres d'attaque

Les paramètres d'attaque désignent les paramètres d'équipement d'injection de fautes en plus du délai d'injection de faute nécessaire à l'exécution d'une attaque donnée. On peut générer facilement les paramètres d'attaque en combinant les résultats précédents et

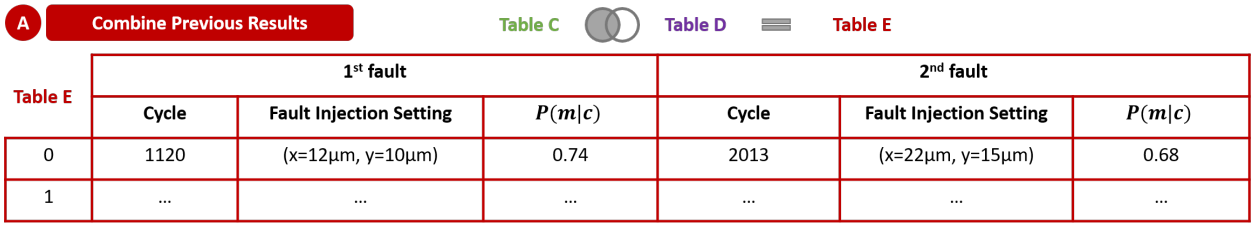


Table E	1st fault			2nd fault		
	Cycle	Fault Injection Setting	$P(m\|c)$	Cycle	Fault Injection Setting	$P(m\|c)$
0	1120	(x=12µm, y=10µm)	0.74	2013	(x=22µm, y=15µm)	0.68
1	...	...	...	...	...	...

FIGURE 4.5 – Combinaison des résultats précédents.

Table F	1st fault			2nd fault		
	Injection Delay	Fault Injection Setting	$P(m\|c)$	Injection Delay	Fault Injection Setting	$P(m\|c)$
0	13280 ns ± 500 ns	(x=12µm, y=10µm)	0.74	25635 ns ± 500 ns	(x=22µm, y=15µm)	0.68
1	...	...	...	...	...	...

FIGURE 4.6 – Conversion des cycles en délai d'injection.

en convertissant approximativement les cycles d'exécution des instructions à fauter en délai d'injection en seconde.

Combinaison des résultats Une table regroupant les paramètres d'attaque est générée à partir des informations précédentes. D'une part, à l'étape d'inférence de modèles de faute spécifiques au microcontrôleur, on a lié modèles de faute et configurations d'équipement. D'autre part, à l'étape d'identification des vulnérabilités avec `CELTIC`, on a mis en évidence les cycles d'exécution d'instruction à fauter ainsi que les modèles de faute conduisant à une vulnérabilité. Plus précisément, la table des paramètres d'attaque (figure 4.5) est obtenue par jointure à gauche sur les modèles de faute de la table de l'inférence de modèles de faute avec la table des attaques réussies simulées avec `CELTIC`.

Conversion des cycles en délai d'injection Les cycles d'exécution des instructions à fauter sont convertis en délais d'injection en seconde (figure 4.6) en utilisant une relation linéaire. L'évaluateur relève expérimentalement le début t_0 et la fin t_1 de l'exécution de référence de l'application. Similairement à Dureuil [Dur16], l'évaluateur détermine le temps moyen exécution d'une instruction α tel que $t_1 - t_0 \approx \alpha c$ avec c le nombre d'instructions de la trace d'exécution de référence.

Cette approximation suppose que l'exécution de chaque instruction nécessite le même nombre de cycles, ce qui est vrai pour la majorité des instructions arithmétiques et logiques, mais pas forcément pour les instructions avec écriture ou lecture mémoire. Également, selon les applications, il n'est pas toujours facile de relever précisément les temps t_0 et t_1. Enfin les modèles ISA ne capturent pas toutes les subtilités du phénomène physique, ce qui induit une autre source d'erreur sur le calcul du délai d'injection. C'est pourquoi, une marge d'erreur arbitraire équivalent à 5% du nombre d'instructions de la trace de référence est rajoutée.

Expérience	µC	Fabricant	Application	Technique d'Injection
1	µC 1	Fabricant A	hVP5	Laser
2	µC 2	Fabricant A	hVP5	Power Glitch
3			hVP5CFI	Power Glitch
4	µC 3	Fabricant B	hVP5	Power Glitch
5			hVP5CFI	Power Glitch

TABLE 4.2 – Description du microcontrôleur, de l'application et de la technique d'injection de faute pour chaque expérience.

4.3.4.2 Sélection des paramètres d'attaque

La sélection des paramètres d'attaque (figure 4.6) est proportionnelle à la probabilité $P(m|c)$. Cette stratégie de sélection est identique à la sélection par tirage à la roulette avec acceptation stochastique (*roulette-wheel selection via stochastic acceptance*) [LL12] que l'on retrouve dans les algorithmes génétiques (voir chapitre 2).

Une fois les paramètres d'attaque sélectionnés, l'équipement est configuré avec les paramètres d'équipement et les fautes sont injectées aux délais d'injection correspondants. Aucune action n'est attendue de la part de l'évaluateur. Lorsqu'une attaque réussie est obtenue, les paramètres d'attaque ainsi que d'éventuelles informations supplémentaires (e.g. l'état final de l'application) sont sauvés dans un ficher de résultats.

4.4 Vue générale des expérimentations

Notre méthodologie a été utilisée sur différentes cibles pour montrer que notre approche s'adapte facilement aux diverses situations rencontrées pendant les évaluations de sécurité. Plus précisément, nous avons injecté des fautes par perturbation laser et par perturbation de la tension d'alimentation, sur trois microcontrôleurs 32-bit et deux implémentations différentes d'un VerifyPIN. Le tableau 4.2 détaille les cibles utilisées pour chaque expérimentation.

Dans les paragraphes suivants, nous présentons succinctement les microcontrôleurs utilisés ainsi que les implémentations évaluées du VerifyPIN. Les expériences 1 et 5 seront détaillées dans la section 4.5. La performance générale de notre méthodologie sera évaluée à la section 4.6 en se basant sur ces 5 expériences.

4.4.0.1 Microcontrôleurs cibles

Tous les microcontrôleurs sélectionnés sont basés sur un cœur ARM Cortex-M4, reposant sur une architecture Harvard ARMv7E-M avec un *pipeline* de trois étages (chargement, décodage, exécution) et prédiction de branchement. Cependant, même s'ils utilisent le même cœur, l'agencement de la puce et les caractéristiques sont différents entre chaque modèle, et

donc ils ne réagissent pas de la même manière aux injections de fautes (cf. tableau 4.7). Le premier microcontrôleur sélectionné (µC 1) est un microcontrôleur dont le cœur est cadencé à 16 Mhz pendant l'expérience. Il a été ouvert par l'arrière pour réaliser des perturbations laser. Le deuxième (µC 2), provenant du même fabricant, est un microcontrôleur ultra basse consommation, cadencé à 48Mhz pendant l'expérience. Enfin le dernier microcontrôleur (µC3) provient d'un fabricant différent. C'est le microcontrôleur le plus rapide de nos expérimentations avec un cœur cadencé à 120 Mhz.

4.4.0.2 Implémentation cibles

Les deux applications évaluées sont des implémentations différentes du VerifyPIN5 du benchmark FISSC [DPP+16], qui est un programme d'authentification avec un code PIN à 4 chiffres. La version originale du VerifyPIN5 utilise plusieurs contre-mesures logicielles, à savoir, des booléens endurcis, du double appel et exécution à temps constant. Pour les deux applications, l'oracle consiste à être authentifié avec un code PIN invalide, sans déclencher une contremesure. Le code est instrumenté pour renvoyer les valeurs des variables critiques à la fin de l'exécution du VerifyPIN.

Notre première implémentation (hVP5) reprend les contre-mesures du VerifyPIN5 mais ajoute des points de contrôles supplémentaires pour se protéger d'un saut de plusieurs instructions. La deuxième implémentation (hVP5CFI) est basée sur notre implémentation hVP5 avec une contremesure additionnelle d'intégrité de flot de contrôle. Cette dernière est générée automatiquement avec l'outil CFI-C de Heydemann et al. [HLB19].

Les traces d'exécution de hVP5CFI sont approximativement 10 fois plus longues que celles de hVP5 ; autour de 200 cycles pour la trace exécution de référence de hVP5 et environ 2000 cycles pour celle de hVP5CFI. Cela rend l'identification et l'exploitation de vulnérabilités à l'injection de faute multiples beaucoup plus difficile sur hVP5CFI.

4.5 Expériences

Dans cette section, nous détaillons les expériences 1 et 5 (tableau 4.2). Ces expériences ont été retenues car elles diffèrent en termes de choix de microcontrôleur, et de la technique d'injection de faute. L'objectif est de montrer que notre méthode s'adapte facilement aux différentes situations qu'un évaluateur peut rencontrer.

4.5.1 Expérience 1 : double injection de faute laser à différentes positions

Notre approche va être utilisée pour identifier et exploiter des vulnérabilités à l'injection de fautes multiples. La particularité de cette expérience est d'utiliser un banc laser double spot pour viser différentes positions sur la puce afin d'induire des sauts d'instruction de différentes longueurs pour contourner l'authentification du hVP5.

4.5.1.1 Banc laser double spot

Notre banc de test pour l'injection de faute laser est constitué de deux diodes laser indépendantes, partageant des caractéristiques identiques, et émettant autour de 1 µm de longueur d'onde. Cette configuration a été inspirée par Selmke et al. [SHS16] ou encore le banc laser D-LMS Alphanov. Chaque faisceau laser traverse la même lentille de focalisation 20x et peut être ajusté indépendamment l'un de l'autre ; nous pouvons régler la puissance de crête de l'impulsion laser, la forme du faisceau et les positions spatiales x, y, dans le champ de vision de la lentille de focalisation.

Enfin, l'illumination laser est déclenchée par un simple signal électrique qui peut être envoyé au bon moment. On peut donc injecter simultanément deux failles précises à des positions temporelles et spatiales différentes ; nous utiliserons cette propriété pour trouver des attaques multi-fautes complexes.

4.5.1.2 Première étape : Inférence de modèles de faute

Les deux lasers du banc partageant les mêmes caractéristiques, la caractérisation est réalisée seulement avec une diode laser. La durée de pulsation du laser est d'environ 60 ns ($\approx$ 1 coup d'horloge) et la puissance-crête du laser est d'environ 1W.

La durée de pulsation et la puissance du laser n'ont pas été optimisés automatiquement car la recherche par quadrillage nous limite à 2-3 dimensions (chapitre 2). Les valeurs retenues sont des valeurs qui avait déjà donné des résultats satisfaisants pour d'autres cibles. D'autres techniques d'optimisation pourraient permettre de gérer plus de dimensions (voir chapitre 6). Par contre, les positions x, y visées avec le laser sur la puce sont balayées automatiquement avec une recherche par quadrillage.

Le code de test utilisé pour la caractérisation est similaire à celui proposé par Proy et al. [PHM+19], détaillé en annexe (Test de caractérisation de fautes T1, tableau A.I).

La partie logique de la mémoire Flash, chargée de l'adressage, est scannée méticuleusement, dans le but de perturber l'étage de chargement et de décodage du *pipeline* dans le but d'induire des changements de flot de contrôle inattendus. Une recherche par quadrillage est utilisée pour trouver les positions avec une probabilité élevée d'induire des résultats fautés (chapitre 2). La partie scannée la plus intéressante de la mémoire Flash se situe dans un rectangle de 150 µm par 200 µm (figure 4.7).

Environ 70.000 fautes ont été injectées pendant 6 heures. Sur ces 70.000 fautes, 12.000 résultats fautés ont été obtenus. La grande majorité des injections fautes ($\approx$ 50.000 fautes) conduisent à des erreurs fatales non récupérables ; c'est-à-dire qu'une exception matérielle est générée, interrompant prématurément l'exécution du test de caractérisation de faute, ou à l'inverse, que le microcontrôleur reste coincé dans une boucle infinie (*timeout*). Le reste des fautes injectées ($\approx$ 4500 fautes) conduisent au résultat attendu.

Une vue agrandie de la zone intéressante de la puce, détaillée à la figure 4.7, permet de mettre en évidence les positions ayant une probabilité de faute élevée, en particulier la région centrale ($x = 1070$ µm, $y = 1250$ µm), avec une probabilité supérieure à 0.9.

En parallèle de la caractérisation, `CELTIC` simule 400.000 injections de faute sur hVP5, générant plus de 100.000 résultats fautés différents en se basant sur l'ensemble de modèles

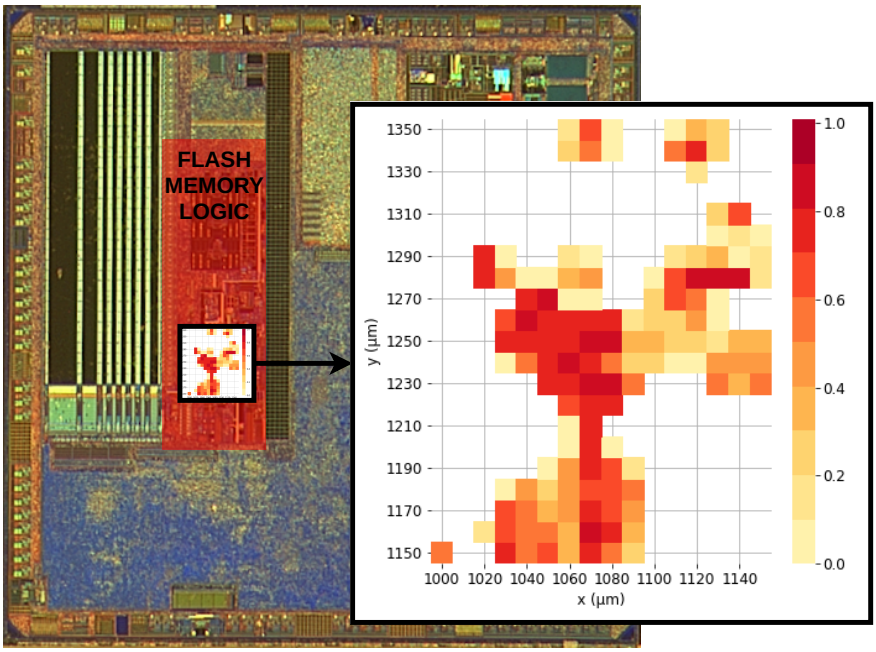


FIGURE 4.7 – Vue arrière du cœur ARM Cortex-M4 du µC1, avec la partie logique de la mémoire flash mis en évidence en rouge, ainsi qu'une vue agrandie des positions ciblées les plus intéressantes. La carte thermique détaille la probabilité d'induire des résultats fautés pour chacune de ces positions.

de fautes par défaut.

Ensuite, les modèles de fautes spécifiques au microcontrôleur µC1 sont inférés à partir des résultats de la caractérisation et de la simulation. Le tableau 4.3 présente les modèles inférés. Parmi les 12.000 résultats fautés obtenus pendant la caractérisation, environ 9.000 d'entre eux peuvent être expliqués avec des modèles de faute grâce à notre approche. Les modèles inférés sont majoritairement des sauts de plusieurs instructions. En particulier, les modèles `16InstructionSkip`, `32InstructionSkip` et `48InstructionSkip` (sous-section 3.2.4) décrivent un saut de 16, 32 et 48 octets respectivement par rapport à l'adresse normalement chargée. Concrètement, cela permet de sauter 4 à 12 instructions de suite.

4.5.1.3 Deuxième étape : Identification des vulnérabilités par simulation

En fixant arbitrairement $P_{threshold} = 0.25$, seuls les trois modèles de saut d'instructions décrits précédemment ont $P(m|c) > P_{threshold}$. Avec ces trois modèles, `CELTIC` ne trouve pas de vulnérabilité à l'injection d'une seule faute, notamment grâce à l'ajout de points de contrôle supplémentaires dans le code de notre version hVP5. Cependant, `CELTIC` identifie plusieurs vulnérabilités à l'injection de double faute, obtenant plus de 400 exécutions fautées de hVP5 contournant le mécanisme d'authentification. Nous allons maintenant, en guise d'exemple, détailler une des attaques trouvées par `CELTIC`.

Le cœur de hVP5 (Listing 4.1) se base sur une boucle qui permet de vérifier si le code PIN à 4 chiffres entré par l'évaluateur est correct. Le principe de l'attaque sélectionnée est

Total	100%	67872
├─ Erreurs Fatales	75%	51172
├─ Résultats Normaux	7%	4576
└─ Résultats Fautés	18%	12124
├─ Résultats Fautés non Couvert	25%	3173
└─ Résultats Fautés Couvert	**75%**	**8951**
├─ `16InstructionSkip`	50%	4497
├─ `48InstructionSkip`	34%	3025
├─ `32InstructionSkip`	10%	887
└─ Autres Modèles	6%	542

TABLE 4.3 – Vue d'ensemble des résultats d'inférence de modèle pour l'expérience 1.

Listing 4.1 Boucle de verification du code PIN de hVP5

```
1:    BOOL diff = BOOL_FALSE;
2:    ...
3:    for (i = 0; i < PINSIZE; i++) {
4:        if (userPIN[i] != cardPIN[i]) {
5:            diff = BOOL_TRUE;
6:        }
7:    }
8:
9:    if (i != PINSIZE) {
10:       countermeasure();
11:   }
```

extrêmement simple. Tout d'abord, il faut sauter au moins 32 octets pour complétement contourner cette boucle, de la ligne 3 à la ligne 11. Cela permet de laisser le booléen `diff` à l'état faux (ligne 1), indiquant qu'il n'y pas de différence entre le code PIN de référence, et celui entré par l'évaluateur, autrement dit que le code PIN entré par l'évaluateur est correct (alors que ce n'est pas le cas). Du fait de la contremesure de double appel, il est important pour réussir l'attaque de sauter une deuxième fois la boucle de vérification.

Pour résumer, cette attaque simulée avec `CELTIC`, consistant à contourner deux fois la boucle de vérification, peut se faire avec le modèle 32InstructionSkip, ou bien le modèle 48InstructionSkip, ou encore une combinaison des deux. Si l'option de combiner deux modèles de faute est rarement considérée pendant les évaluations de sécurité, sous prétexte d'être difficilement réalisable en pratique, l'utilisation de plusieurs sauts d'instructions différents a été étudiée théoriquement par Penau et al. [PCHR20] pour réaliser des attaques complexes. Nous allons montrer que notre approche permet de facilement et rapidement réaliser expérimentalement ces attaques complexes.

	Laser 1 (`48InstructionSkip`)			Laser 2 (`32InstructionSkip`)		
	Position (µm)	Délai d'Injection (ns)	$P(m\|c)$	Position (µm)	Délai d'Injection (ns)	$P(m\|c)$
#0	x=1050, y=1270	4468 ± 700	0.74	x=1060, y=1240	9143 ± 700	0.68
#1	...	...	...	...	...	...

TABLE 4.4 – Table des paramètres d'attaque générés pour le banc laser double spot d'essai.

4.5.1.4 Troisième étape : Exploitation des vulnérabilités

Les deux diodes laser indépendantes de notre banc sont réglées pour viser différentes positions de la puce, qui d'après l'étape d'inférence de modèles de faute, induisent les modèles 48InstructionSkip et 32InstructionSkip. À notre connaissance, c'est la première fois que deux diodes laser indépendantes focalisent différentes positions de la puce à travers le même faisceau optique, dans le but d'induire sciemment différents sauts d'instructions. À l'inverse, Selmke et al. [SHS16] injectent les même fautes à différentes positions pour contrecarrer des contre-mesures basées sur des mécanismes de redondance.

Plus précisément, pour réaliser l'attaque simulée avec `CELTIC` avec combinaison des modèles de faute, le laser 1 est configuré pour viser une position liée au modèle 48InstructionSkip, tandis que le laser vise une position connue pour induire le modèle 32InstructionSkip (figure 4.8). Des exemples de paramètres d'attaques générés automatiquement avec notre approche sont présentés au tableau 4.4. À partir de ces paramètres d'attaque, il est possible de contourner l'authentification de hVP5 en quelques essais seulement (i.e. temps d'exploitation < 1 min).

4.5.2 Expérience 5 : corruption du cache d'instruction avec deux glitches

L'objectif de cette expérience est d'appliquer notre méthodologie sur un code plus long et plus complexe, tout en utilisant un équipement d'injection de faute moins onéreux et plus facile d'accès qu'un banc laser double spot. Nous arrivons à identifier et exploiter des vulnérabilités à l'injection de fautes multiples sur une application protégée par 5 contre-mesures logicielles différentes. Notamment, nous injectons plusieurs fautes pour corrompre le cache d'instruction du microcontrôleur ciblé à plusieurs reprises, en utilisant la `GLITCH STATION` (chapitre 3).

4.5.2.1 Première étape : Inférence de modèles de faute

Pendant la caractérisation, pour optimiser les paramètres d'équipement de la `GLITCH STATION`, nous utilisons un algorithme génétique, car une recherche par quadrillage n'est pas envisageable, comme détaillé au chapitre 3.

Comme pour l'expérience précédente, le code de test utilisé pour la caractérisation est similaire à celui proposé par Proy et al. [PHM+19], détaillé en annexe (Test de caractérisation

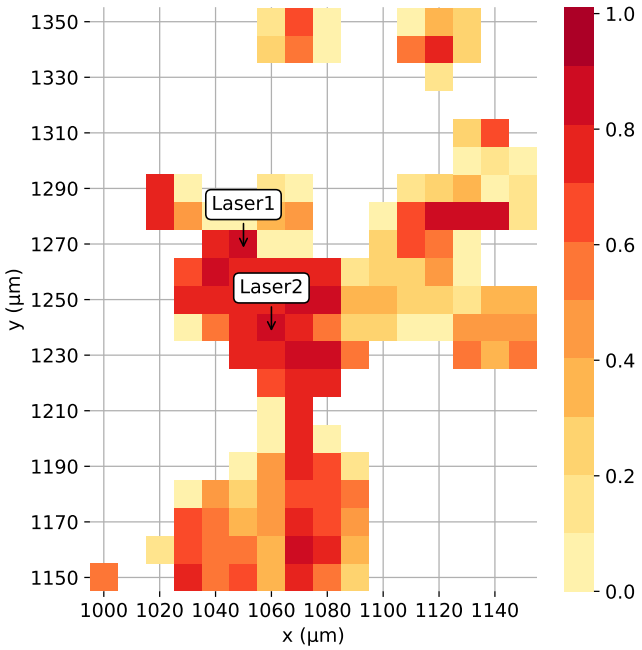


FIGURE 4.8 – Exemple de couple de positions contournant l'authentification de hVP5. Le premier laser vise une position liée au modèle `48InstructionSkip` tandis que le le deuxième laser vise une position liée au modèle `32InstructionSkip`.

de fautes T1, tableau A.I).

Environ 200.000 fautes sont injectées ($\approx$ 12 heures) et un peu moins de 70.000 résultats fautés sont obtenus. Très peu de résultats normaux sont observés (seulement 1% des fautes injectées). À l'inverse, une grande majorité des fautes conduisent à une erreur fatale (62% des fautes injectées). La figure 4.9 présente une forme de glitch avec probabilité de faute élevée (> 0.9) identifiée pendant la caractérisation. En parallèle, `CELTIC` permet de simuler 400.000 injections de fautes et générer plus de 100.000 résultats fautés, similairement à l'expérience précédente.

L'inférence de modèles de faute met en évidence qu'une grande partie des fautes induisent des corruptions du cache d'instruction ($\approx$ 90%). Le modèle `4CacheCorruption` (sous-section 3.2.4), déjà mis évidence par Rivière et al [RNR+15] en utilisant un équipement d'injection EM, permet de rejouer les 4 dernières instructions exécutées et de sauter les 4 suivantes.

4.5.2.2 Deuxième étape : Identification des vulnérabilités par simulation

`4CacheCorruption` est l'unique modèle avec une probabilité $P(m|c)$ significative et sera par conséquent le seul simulé par `CELTIC` pour identifier de potentielles vulnérabilités dans hVP5CFI. Comme pour l'expérience précédente, `CELTIC` ne trouve aucune vulnérabilité à l'injection d'une seule faute. Malgré la contre-mesure additionnelle d'intégrité de flot de contrôle, 64 vulnérabilités à l'injection de double faute sont identifiées comme pouvant contourner le

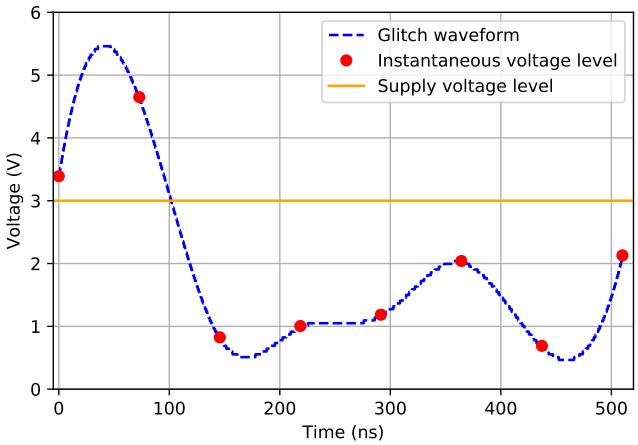


FIGURE 4.9 – Exemple d'une forme de glitch générée avec la `GLITCH STATION`.

Total	100%	185517
├─ Erreurs Fatales	62%	115011
├─ Résultats Normaux	1%	2046
└─ Résultats Fautés	37%	68460
├─ Résultats Fautés non Couvert	10%	6604
└─ Résultats Fautés Couvert	**90%**	**61846**
├─ `4CacheCorruption`	99%	61818
└─ Autres Modèles	1%	38

TABLE 4.5 – Vue d'ensemble des résultats d'inférence de modèle pour l'expérience 5.

mécanisme d'authentification de hVP5CFI. En particulier, une attaque, que nous détaillerons, a retenu notre attention en exploitant une faiblesse dans la contre-mesure d'intégrité de flot de contrôle automatique de l'outil CFI-C [HLB19].

Une des sections les plus critiques (figure 4.10) de l'application hVP5CFI est le dernier branchement conditionnel, après la boucle de vérification du code PIN entré par l'évaluateur, qui teste l'état de la variable booléenne `diff`. Comme pour hVP5, pour être authentifié, il ne doit pas avoir de différence avec le code PIN attendu (`diff == BOOL_FALSE`). Le code contient de nombreuses macros générées automatiquement pour l'outil CFI-C pour garantir l'intégrité de flot de contrôle. Trois de ces macros (explicitées dans [HLB19]) sont soulignées en vert dans la figure 4.10 :

- La macro `INIT(B,1)` initialise une variable booléenne supplémentaire B, par défaut à l'état *vrai*, et qui sera chargée de contenir la valeur de l'expression logique `diff == BOOL_FALSE`. La variable `B` peut être vue comme un garde-fou et sera utilisée ultérieurement pour vérifier l'intégrité du flot de contrôle.

- La macro `INCR_COND` met à jour la variable `B` avec la valeur de l'expression logique évaluée par le branchement conditionnel (ici `diff == BOOL_FALSE`). Pour être authen-

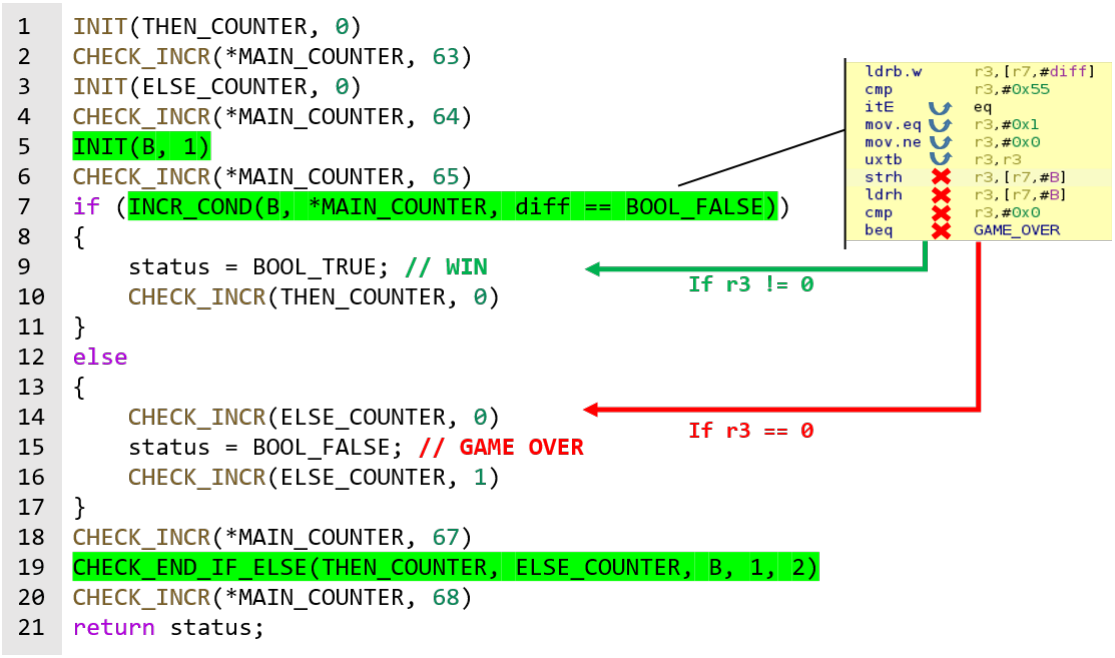


FIGURE 4.10 – Section critique du test d'authentification du hVP5CFI. Avec le modèle `4CacheCorruption`, il est possible d'empêcher la mise à jour l'état de la `B` et passer le test d'authentification.

tifié, la branche `THEN` doit être prise, de manière à valider la vérification du code PIN (`status = BOOL_TRUE`).

- La macro `CHECK_END_IF_ELSE` détecte les incohérences entre le chemin pris et celui attendu, notamment grâce à la variable `B`. Par exemple si la branche `THEN` est prise alors que la variable `B` est fausse ($B = 0$), et donc que l'expression logique (1) de la macro `CHECK_END_IF_ELSE` est *fausse*, une contre-mesure désactivant le microcontrôleur est activée.

```
THEN_COUNTER == x && ELSE_COUNTER == 0 && B      (1)
```

Pour résumer, pour forcer l'authentification, il faut réussir à prendre la branche `THEN` tout en s'assurant de la cohérence avec la variable `B`. Cependant, un lecteur attentif aura sans doute remarqué que la variable `B` de l'expression (1) n'est pas testée correctement, et que n'importe quelle valeur, hormis 0, est valide. De plus, la variable B est par défaut initialisée à vraie ($B = 1$).

Avec ces éléments, deux chemins d'attaque sont identifiés avec `CELTIC` pour contourner le mécanisme d'intégrité de flot de contrôle, au niveau de la macro `INCR_COND` (figure 4.10) :

- Sans rentrer dans les détails, le registre `R3` est critique dans la macro `INCR_COND`. En forçant le registre `R3` à une valeur différente de zéro, cela permet de s'authentifier sans déclencher une contre-mesure.

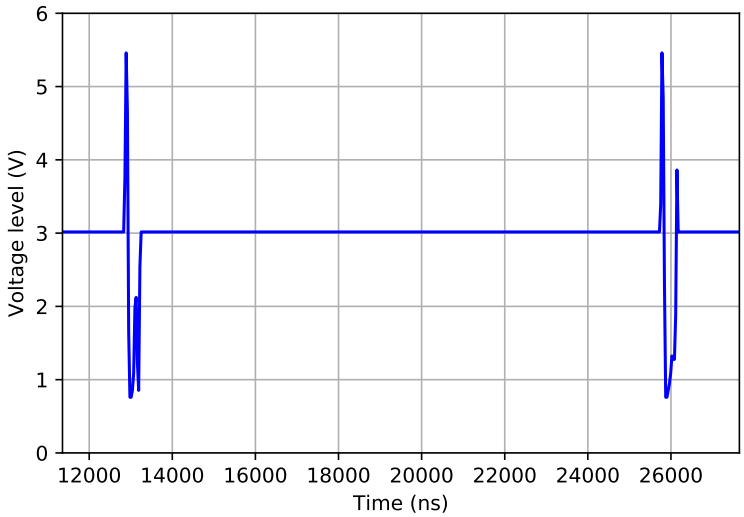


FIGURE 4.11 – Exemple de double glitch contournant l'authentification du hVP5CFI.

- L'alternative retenue est de corrompre le cache d'instruction pour sauter les 4 dernières instructions après l'évaluation de l'expression `diff == BOOL_FALSE`. De cette manière, la variable `B` n'est pas mise à jour et on saute directement dans la branche `THEN`.

Comme hVP5CFI utilise aussi une contre-mesure de double appel, il faut réussir à corrompre le cache d'instruction deux fois de suite pour s'authentifier sans déclencher de contre-mesure. Pour résumer, avec le modèle de faute `4CacheCorruption`, `CELTIC` a identifié des vulnérabilités à l'injection de double faute, difficilement identifiables manuellement au vu de la complexité du code.

4.5.2.3 Troisième étape : Exploitation des vulnérabilités

Si la technique d'injection est différente de l'expérience précédente, la méthodologie reste la même. Les résultats de l'inférence de modèle de fautes sont combinés avec les attaques simulées avec `CELTIC` pour générer le tableau 4.6 des paramètres d'attaque. figure 4.11 présente la forme du double glitch correspondant au premier paramètre d'attaque du tableau 4.6. Ce double glitch permet de corrompre le cache d'instruction du microcontrôleur µC3 deux fois de suite pour réussir l'attaque présentée précédemment. Encore une fois, en seulement quelques tentatives on arrive à réussir l'attaque (soit <1min de temps d'exploitation) et être authentifié, malgré les 5 contre-mesures logicielles.

4.6 Évaluation de la méthodologie

Si les deux expériences présentées dans la section précédente montre que notre méthodologie permet d'identifier et d'exploiter des vulnérabilités à l'injection de fautes multiples,

	1er Glitch (4CacheCorruption)				2ème Glitch (4CacheCorruption)			
	Niveaux de Tension (V)	Durée (ns)	Délai d'Injection (ns)	$P(m\|c)$	Niveaux de Tension (V)	Durée (ns)	Délai d'Injection (ns)	$P(m\|c)$
#0	[3.8, 5.5, 0.8, 0.8, 1.0, 2.3, 1.0, 2.5]	390	13280 ± 500	0.89	[3.4, 5.2, 0.9, 0.8 1.0, 1.4, 1.3, 3.9]	420	25635 ± 500	0.91
#1	...	...	...	...	...	...	...	...

TABLE 4.6 – Table des paramètres d'attaque générés pour la GLITCH STATION.

µC	Technique d'Injection	Modèle Représentatif	Taux de Couverture
µC 1	Laser	16InstructionSkip	75%
µC 2	Power Glitch	8InstructionSkip	64%
µC 3	Power Glitch	4CacheCorruption	90%

TABLE 4.7 – Le taux de couverture et le modèle le plus représentatif selon le microcontrôleur évalué et la technique d'injection de faute.

il est important d'évaluer notre approche sur plusieurs critères, à savoir, 1) la couverture de fautes, 2) la détection de vulnérabilités et 3) la vitesse d'exploitation.

4.6.1 Couverture de fautes

Notre méthodologie repose en grande partie sur les résultats de l'étape d'inférence de modèles de faute. Par conséquent, nous devons nous assurer que la majorité des résultats fautés observés sont couverts par des modèles pour différents microcontrôleurs et techniques d'injection de faute. Pour chaque microcontrôleur du tableau 4.2, nous avons évalué la proportion des résultats fautés qui peuvent être expliqués avec l'ensemble de modèles de faute par défaut (tableau 3.1).

Le tableau 4.7 présente le taux de couverture global et le modèle le plus représentatif pour chaque microcontrôleur évalué. De manière générale, notre approche permet d'expliquer 76% des résultats fautés observés en moyenne. Malgré l'utilisation de technique d'injection différentes, la majorité des fautes induisent des modifications du flot de contrôle, en particulier des sauts d'instruction.

Des modèles de fautes plus complexes sont nécessaires pour comprendre les résultats fautés non couverts. Ces modèles plus complexes, comme par exemple ceux présentés par Laurent et al. [LBD+19] décrivant des fautes pendant la phase de *writeback* d'un cœur RISC-V, ne sont pas facilement implémentables dans CELTIC. Les limitations seront abordées plus en détail dans la section 4.7.

4.6.2 Détection de vulnérabilités et vitesse d'exploitation

Pour évaluer la détection de vulnérabilités et la vitesse d'exploitation, nous comparons notre méthodologie avec une approche communément utilisée lors d'évaluations de sécu-

µC	Technique d'Injection	Application	Taux de Détection
µC 1	Laser	hVP5	51%
µC 2	Power Glitch	hVP5	32%
		hVP5CFI	17%
µC 3	Power Glitch	hVP5	38%
		hVP5CFI	100%

TABLE 4.8 – Taux de détection de vulnérabilité en fonction de l'application et du microcontrôleur évalués.

rité. Concernant l'évaluation de la détection de vulnérabilité, l'objectif est de vérifier si les résultats simulés avec `CELTIC`, en se basant sur les modèles de fautes inférés, permettent d'identifier toutes les vulnérabilités *réelles* sur le microcontrôleur. Quand à l'évaluation de la vitesse d'exploitation, le but est de s'assurer que notre méthodologie permet de réussir une attaque sur le banc plus rapidement qu'avec une approche plus classique. Ainsi, les deux approches évaluées sont les suivantes :

- Approche basée sur la caractérisation seule (**Approche A**) : l'évaluateur connait les paramètres d'équipement avec une probabilité de faute élevée pour le microcontrôleur évalué grâce à une caractérisation. Cependant, contrairement à notre approche, l'évaluateur n'a aucune connaissance préalable des sections critiques de l'application, l'obligeant à couvrir entièrement cette dernière.
- Notre approche (**Approche B**) : l'évaluateur connait les meilleurs paramètres d'attaque (paramètres d'équipement et délais d'injection) ce qui permet de réduire considérablement l'espace de recherche.

Notez que l'approche naïve, consistant à trouver au hasard à la fois les délais d'injection et les paramètres d'équipement, ne permet pas de réussir une attaque dans un délai raisonnable.

4.6.2.1 Détection de vulnérabilités

Dans ce paragraphe, nous vérifions que nous pouvons retrouver la majorité des vulnérabilités identifiées avec l'approche classique en utilisant notre méthodologie. Plus précisément, on évalue le ratio de détection de vulnérabilités entre l'approche basée sur la caractérisation seule (**Approche A**) et notre approche (**Approche B**). Le tableau 4.8 détaille le ratio de détection pour chacune des expériences. En moyenne, on parvient à retrouver 48% des vulnérabilités identifiées avec l'approche A.

Les figure 4.12 et figure 4.13 présentent respectivement le meilleur et le pire scénario possible. Sur ces figures, on retrouve 1) les vulnérabilités identifiées avec l'approche A symbolisés par les points verts, 2) la zone couverte par l'approche A en jaune et 3) la zone couverte avec notre approche en orange. Les deux axes correspondent au délai d'injection pour la première et la deuxième faute. Si pour expérience 5 (figure 4.12) notre approche

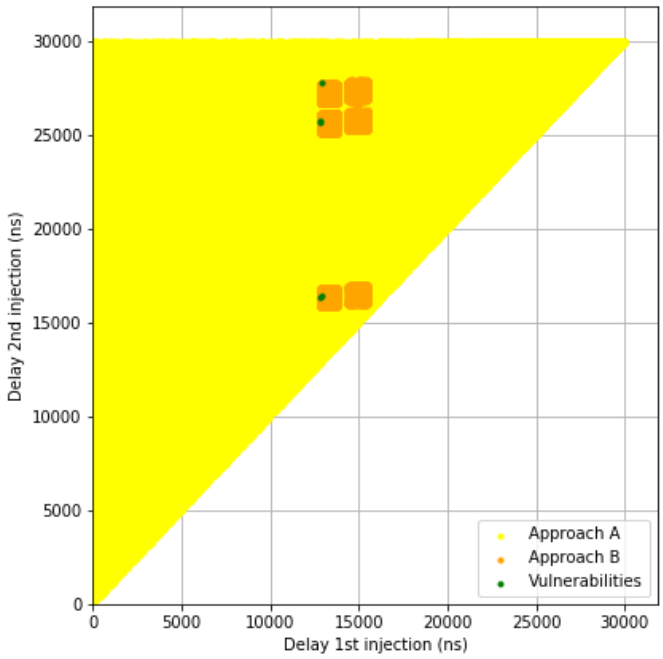


FIGURE 4.12 – Le meilleur scenario possible. Notre approche a détecté 100% des vulnérabilités d'injection multi-fautes identifiées avec l'approche basée sur la caractérisation pour le microcontrôleur µC3 et l'application hVP5CFI.

permet de couvrir toutes les vulnérabilités identifiées, pour l'expérience 3 non développée ici (figure 4.13), seulement 17% d'entre elles sont identifiées. Les raisons pouvant expliquer ces différences observées entre les expériences seront détaillées dans la section 4.7.

4.6.2.2 Vitesse d'exploitation

Pour évaluer la vitesse d'exploitation des vulnérabilités de notre méthodologie, nous comparons la temps moyen d'exploitation d'une vulnérabilité à l'injection de fautes multiples avec l'approche basée sur une caractérisation seule (**Approche A**) et notre approche (**Approche B**).

Plus précisément, seul le temps d'exploitation est comptabilisé, c'est-à-dire le temps passé sur l'équipement d'injection dans le but de réussir un scénario d'attaque. Le temps de caractérisation (identique pour les deux approches) et le temps de simulation avec `CELTIC` (uniquement pour notre approche) ne sont pas comptabilisés. Le temps de simulation, rajouté par de notre approche, varie de quelques minutes pour l'application hVP5 à plusieurs heures pour l'application hVP5CFI sur un PC de bureau. Nous estimons que le temps de simulation sur un PC de bureau n'est pas comparable au temps d'exploitation sur un équipement d'injection de faute car lorsqu'il y a plusieurs évaluations de sécurité en cours, trouver un PC de bureau libre est souvent plus facile qu'un banc laser double spot. De plus, il n'y a aucun risque d'endommager le dispositif lors de la simulation d'injection de faute.

Le tableau 4.9 détaille le temps moyen et maximal d'exploitation pour chaque expérience,

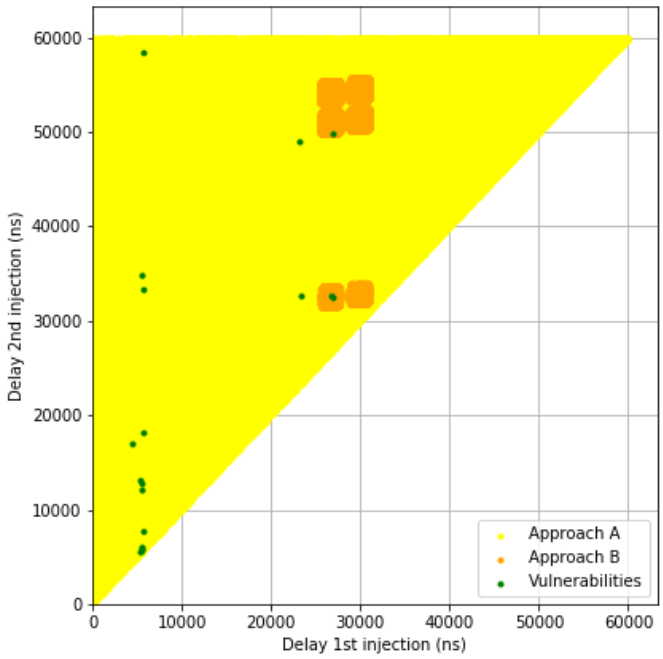


FIGURE 4.13 – Le pire scenario possible. Notre approche a détecté 17% des vulnérabilités d'injection multi-fautes identifiées avec l'approche basée sur la caractérisation pour le microcontrôleur µC2 et l'application hVP5CFI.

en fonction de l'approche choisie. Chaque expérience est répétée en moyenne 50 fois pour obtenir des résultats représentatifs. De manière générale, notre approche (**Approche B**) est 10 fois plus rapide en moyenne pour exploiter une vulnérabilité à l'injection de faute multiple que l'approche classique (**Approche A**). En particulier, notre approche est 20 fois plus rapide en moyenne que l'approche classique avec le microcontrôleur µC3, mais seulement 2 fois plus rapide en moyenne avec le microcontrôleur µC2. Nous détaillerons les raisons conduisant à ces différences de performance mais aussi les possibles pistes d'amélioration de notre méthodologie dans ce qui suit.

4.7 Amélioration de notre méthodologie

L'objectif de cette section est de comprendre les raisons des disparités des résultats observées lors de l'évaluation de notre méthodologie. Après avoir exposé les différentes limitations observées précédemment, nous reviendrons sur les possibles pistes d'améliorations de notre méthodologie.

4.7.1 Les limitations de notre méthodologie

Que ce soit durant les étapes d'inférence de modèles de faute, d'identification de vulnérabilités, ou encore d'exploitation de vulnérabilités, plusieurs limitations expliquent les

Expérience	µC	Technique d'Injection	Application	Approche A		Approche B	
				Temps Moyen	Temps Max	Temps Moyen	Temps Max
1	µC 1	Laser	hVP5	14min	2h36min	**4min**	**31min**
2	µC 2	Power Glitch	hVP5	1min20sec	5min	**1min**	**5min**
3			hVP5CFI	3h12min	7h30min	**1h30min**	**4h15min**
4	µC 3	Power Glitch	hVP5	21min	1h53min	**3min**	**20min**
5			hVP5CFI	8h03min	17h23min	**13min**	**1h02min**

TABLE 4.9 – Le temps d'exploitation maximum et moyen pour réussir une attaque multi-faute, avec l'approche basée sur la caractérisation seule (**Approche A**) et avec notre approche (**Approche B**), en fonction du microcontrôleur ciblé, de l'application évaluée, et de la technique d'injection de faute utilisée.

disparités observées entre les expériences, notamment pendant la détection de vulnérabilités ou bien la vitesse d'exploitation.

- 1ère limitation, *les modèles de faute utilisés à l'étape d'inférence n'expliquent pas tous les résultats fautés observés*. Certains fautes complexes ne sont pas couvertes par nos modèles. Or, la couverture de fautes influence la détection de vulnérabilités. Par exemple, le microcontrôleur µC2 de l'expérience 3 a le taux de couverture le plus faible du benchmark (64%, tableau 4.7). Ainsi, ces fautes non couvertes sont potentiellement à l'origine des vulnérabilités non détectées par notre méthodologie lors de l'expérience 3.

- 2ème limitation, *les tests de caractérisation de faute ne propagent pas tous les effets pouvant se produire sur le microcontrôleur*. En effet, ces derniers ne représentent pas toute la complexité de l'application cible, ainsi certaines fautes peuvent potentiellement passées sous le radar lors de la caractérisation.

- 3ème limitation, *on ne simule pas tous les modèles spécifiques inférés*. À l'étape d'identification des vulnérabilités de notre méthodologie, on introduit un compromis entre vitesse et couverture en ne simulant seulement qu'une partie des modèles de faute inférés. Si cela permet de réduire le temps de simulation et augmenter la vraisemblance des attaques simulées avec `CELTIC`, nous pouvons potentiellement passer à coté de vulnérabilités comme nous ne simulons qu'une partie des modèles de faute inférés.

- 4ème limitation, *`CELTIC` n'émule pas tous les mécanismes de la microarchitecture ciblée*. Il existe toujours un compromis, lors de la conception d'un émulateur, entre contrôle, rapidité et précision. `CELTIC` prioritise le contrôle et la rapidité au détriment de la précision [Dur16]. `CELTIC` suppose que les instructions durent le même nombre de cycle, ne fait pas de prédiction de branchement et n'émule pas le fonctionnement du *pipeline* du microcontrôleur ciblé. Toutes ces approximations vont se répercuter lors du calcul des délais d'injection.

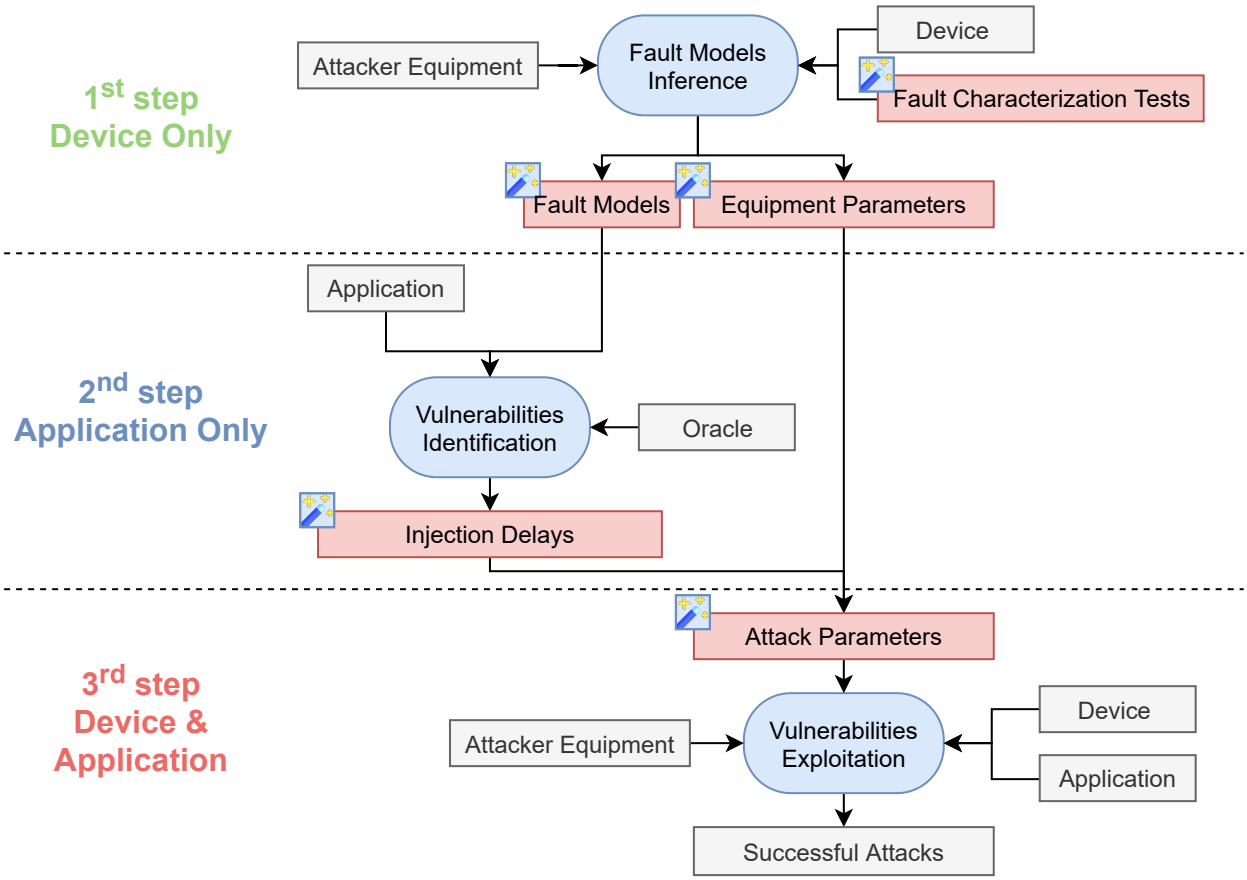


FIGURE 4.14 – Les points critiques de notre approche sujet à amélioration.

- 5ème limitation, *les modèles ISA ne capturent pas toute la complexité de l'effet d'une faute*. La représentation des effets des fautes au niveau ISA ne permet pas de tenir compte de toutes les subtilités de la perturbation au niveau physique, ce qui impacte également le calcul des délais d'injection mais aussi la fidélité de l'effet décrit par le modèle.

- 6ème limitation, *mesurer précisément le début et la fin de l'application ciblée peut s'avérer complexe*. Enfin à l'étape d'exploitation des vulnérabilités, des erreurs de mesure du début t_0 et de la fin t_1 de l'exécution de référence de l'application faussent le calcul des délais d'injection.

4.7.2 Pistes d'amélioration

Pour pallier ces limitations, nous avons identifié plusieurs pistes pour améliorer notre méthodologie, dans le but d'augmenter le taux de couverture de fautes et de détection de vulnérabilités. Ces améliorations concernent directement ou indirectement les éléments mis en évidence sur la figure 4.14. Pour couvrir plus de fautes, inférer plus de modèles spécifiques, trouver plus de paramètres d'attaques optimaux et enfin calculer des délais d'injection plus

précis, nous proposons plusieurs pistes d'amélioration, à savoir :

- L'utilisation de meilleurs tests de caractérisation de fautes pour propager toujours plus d'effets et mieux comprendre le comportement du microcontrôleur ciblé en réponse aux injections de faute.
- L'utilisation de nouvelles techniques d'optimisation pour accélérer l'exploration de l'espace des paramètres dans le but de trouver de meilleurs paramètres d'équipement.
- La combinaison de modèles existant pour former de nouveaux modèles plus complexes dans le but de couvrir plus de fautes, similairement à Alshaer et al. [ACD+21].
- La réutilisation et mutualisation des calculs précédents pour accélérer les simulations d'injection de faute multiples.
- L'émulation complète du *pipeline* et prise en compte du nombre réel de chaque instruction dans `CELTIC` pour augmenter la précision des délais d'injection calculés.
- L'analyse de la consommation ou d'émanations électromagnétiques dans le but d'améliorer la synchronisation avec la cible pour calculer des délais d'injection plus précis.

Parmi ces pistes d'amélioration possibles, nous avons choisi de porter nos efforts sur la formalisation des tests de caractérisation de fautes (chapitre 5) et sur l'application de nouvelles techniques d'optimisation inédites dans le domaine de l'injection de faute (chapitre 6).

4.8 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle méthodologie pour identifier et exploiter des vulnérabilités à l'injection de fautes multiples. La particularité de notre méthodologie est d'être versatile, automatisée et de bout-en-bout, ce qui permet à notre approche d'être facilement adaptable aux situations rencontrées lors d'évaluations de sécurité.

En particulier, nous avons conduit plusieurs expériences avec différents microcontrôleurs, applications et techniques d'injection de faute. Nous avons constaté que notre méthodologie permet d'exploiter des vulnérabilités 10 fois plus rapidement en moyenne qu'une approche uniquement basée sur une caractérisation.

Le cœur de notre méthodologie repose sur une évaluation séparée du microcontrôleur et de l'application. Dans un premier temps, avec le microcontrôleur uniquement, les modèles de fautes spécifiques ainsi que les paramètres d'équipement optimaux sont identifiés en combinant les résultats d'une caractérisation de fautes avec les résultats d'une simulation d'injection de faute. Dans un deuxième temps, avec l'application uniquement, des chemins d'attaque multi-fautes sont identifiés. Malgré l'explosion combinatoire inhérente à l'injection de fautes multiples, cela est rendu possible en limitant la simulation aux modèles spécifiques les plus probables. Enfin, les paramètres d'attaque sont générés automatiquement avec les informations obtenues précédemment, ce qui permet de faciliter l'exploitation des vulnérabilités identifiées.

De plus, l'autre originalité de notre approche est de pouvoir simuler des injections multi-fautes en utilisant différents modèles de faute pour chaque injection, dans le but d'explorer de nouveaux chemins d'attaque. Par ailleurs, en pratique, nous avons réalisé une double injection de faute par perturbation laser à différentes positions de la puce, pour induire de manière contrôlée des sauts d'instructions de différentes longueurs.

La possibilité d'élaborer plus facilement des attaques multi-fautes complexes, remet en question la fiabilité des contre-mesures logicielles seules pour protéger un microcontrôleur contre des attaques par fautes de plus en plus puissantes.

Toutefois, cela est à nuancer car plusieurs limitations pouvant entraver le bon déroulement de notre méthodologie ont été identifiées, même si plusieurs pistes d'amélioration sont envisagées. Parmi elles, l'amélioration de la caractérisation nous semble prioritaire car cette étape affecte grandement le reste de la méthodologie. C'est pourquoi nous formaliserons la conception de tests de caractérisation de fautes dans le chapitre 5 et étudierons de nouvelles techniques d'optimisation des paramètres au chapitre 6 dans le but d'améliorer l'identification et l'exploitation de vulnérabilité.

Chapitre 5

Vers de meilleurs tests de caractérisation de fautes

Sommaire

5.1 Introduction

Dans le chapitre précédent, différentes pistes ont été proposées pour améliorer notre méthodologie, dont l'utilisation de meilleurs tests de caractérisation dans le but de trouver plus facilement des modèles de faute spécifiques. Néanmoins, il n'existe pas de critères objectifs pour juger de la qualité et de la pertinence d'un test. C'est pourquoi, dans ce chapitre, nous allons définir et étudier les critères de conception des tests de caractérisation de fautes afin de mieux comprendre les forces et faiblesses de ces derniers.

Dans ce sens, trois métriques sont proposées, se basant elles-mêmes sur deux propriétés importantes que nous définissons, à savoir, la *propagation* et la *discrimination*. Ces différentes métriques sont utilisées pour comparer les performances de plusieurs tests classiques issus de la littérature, afin d'identifier les critères de conception les plus importants selon le modèle de faute. À partir de ces observations, nous détaillons une approche générale pour concevoir les meilleurs tests de caractérisation de fautes selon nos métriques. Enfin, ces tests seront utilisés pour identifier rapidement les paramètres d'équipement optimaux pour notre `GLITCH STATION` avec un microcontrôleur Cortex-M4 32-bit.

Dans la suite de ce chapitre, nous reviendrons sur les principales motivations dernière cette démarche à la section 5.2. Les propriétés et métriques que nous proposons seront définies à la section 5.3. Ensuite, à la section 5.4, nous évaluerons, à l'aide de ces métriques, plusieurs tests de caractérisation de fautes issus de la littérature puis nous détaillons, à la section 5.5, notre approche pour concevoir de meilleurs tests. Enfin, à la section section 5.6, nous présentons un cas d'utilisation pratique de nos tests.

5.2 Motivations

5.2.1 Contexte

Le premier constat lorsque l'on souhaite utiliser des tests de caractérisation de fautes, est de remarquer la grande diversité de ces derniers dans la littérature. Néanmoins, en regardant de plus près, on peut retrouver certaines similarités entre ces tests comme la structure, ou encore les instructions utilisées. L'accent est généralement mis sur la facilité d'utilisation, c'est-à-dire pouvoir diagnostiquer facilement l'effet de la faute injectée. De plus, comme déjà détaillé au chapitre 2, ces tests sont construits empiriquement, sauf rares exceptions (e.g. [TSW16]). Ainsi, les tests proposés sont rarement optimisés en fonction de critères objectifs, causant plusieurs complications.

D'une part, l'une des premières difficultés rencontrées est de comprendre les motivations derrière chacun de ces tests, ainsi que les hypothèses de modèles de faute prises en compte lors de leurs conception. D'autre part, il peut être difficile de choisir le ou les tests à utiliser, car aucune étude à ce jour ne compare objectivement ces derniers. On peut s'interroger alors sur le niveau de couverture de fautes de ces tests mais aussi de leur pertinence, du fait qu'il soit difficile pour une situation donnée de choisir un test plutôt qu'un autre. D'autant plus lorsque les effets des fautes injectées sur le microcontrôleur ciblé ne correspondent pas aux hypothèses des modèles de faute choisies lors de la conception. Ainsi, Les effets des fautes injectées peuvent être entièrement masqués pendant l'exécution du test. Par exemple, Trouchkine et al. [TBC19] proposent des tests de caractérisation idempotents, y compris pour observer des corruptions d'instruction (lors du chargement, du décodage, ou bien de l'exécution). Toutefois, nous verrons dans la section 5.4, que les tests idempotents sont particulièrement peu performants pour propager ce type d'effet de faute.

Plus généralement, nous montrerons que les tests de caractérisation doivent vérifier les propriétés de *propagation* et de *discrimination* (détaillées en section 5.3) en fonction des modèles de faute supposés. Il n'existe pas encore de directives générales pour concevoir des tests de caractérisation de fautes, d'où les nombreuses conceptions différentes proposées dans la littérature. Ainsi, une meilleure façon de concevoir des tests serait de vérifier et valider systématiquement leur propagation et leur discrimination ,avec une simulation d'injection de faute.

5.2.2 Contributions

Nous proposons des directives générales pour aider à la conception de tests de caractérisation de fautes optimisés pour un large éventail de modèles de faute, sur la base de la première étude approfondie des test de caractérisation de fautes les plus populaires. Ensuite, à l'aide de ces directives générales et `CELTIC`, nous générons des tests de caractérisation de fautes optimaux, spécialisés pour un modèle de faute particulier. Ces tests spécialisés sont alors combinés en un ensemble minimal de tests pour couvrir des corruptions d'instruction, de registre et de mémoire. Ainsi, nos contributions sont les suivantes :

- Proposition de métriques pour évaluer la performance des tests de caractérisation de fautes, selon les modèles de faute supposés.
- Évaluation de tests de caractérisation de fautes issus de la littérature pour mettre en évidence l'impact des choix de conception sur les performances, avec les métriques proposées.
- Proposition de directives générales pour aider à la conception de tests de caractérisation de fautes selon les modèles de faute supposés.
- Proposition de tests optimisés de caractérisation de fautes spécialisé pour un modèle de faute particulier.
- Proposition d'un ensemble minimal de tests pour couvrir de nombreux modèles de faute.

5.3 Propriétés et métriques

Dans cette section, nous définirons deux propriétés pertinentes pour les tests de caractérisation de fautes, la *propagation* et la *discrimination*. Ensuite, des métriques de performances seront dérivées de ces propriétés.

Comme détaillé dans le chapitre 2, le but principal d'un test de caractérisation de fautes est de récupérer le plus d'informations possibles sur le comportement du microcontrôleur ciblé en présence de fautes. Généralement, seul le résultat final, un instantané de l'état du programme à la fin de l'exécution du test, peut être observé. Par conséquent, un test de caractérisation de fautes doit *propager* les effets des fautes sur le microcontrôleur jusqu'à la fin de la séquence d'instruction I. L'effet de faute ne doit pas être masqué lors de l'exécution des instructions dans I, sinon les résultats ne seront pas représentatifs du comportement fauté du microcontrôleur. De plus, de manière à comprendre plus facilement les effets des fautes ou de valider des hypothèses de modèles de faute, chaque état final doit être caractéristique d'un effet de faute particulier. Autrement dit, il ne faut pas qu'un état final soit le résultat de plusieurs effets de fautes différents. Nous dirons que le test de caractérisation de faute doit *discriminer* les différents effets induits par les injections de fautes. Dans ce qui suit, nous formalisons les propriétés de propagation et de discrimination des tests de caractérisation de fautes.

Definition 1 (Propagation) *Soit $s_I^\star$ l'état final référence de I, c'est-à-dire l'état final pour une exécution de I sans faute. Soit S_I^m l'ensemble des états finaux des exécutions de I fautées selon le modèle de faute m. I assure une propriété de propagation vis-à-vis d'un ensemble de modèles de faute M si :*

$$\forall m \in M, s_I^\star \notin S_I^m$$

Definition 2 (Discrimination) *Soit S_I^m l'ensemble des états finaux des exécutions de I fautées selon le modèle de faute m. I assure une propriété de discrimination vis-à-vis d'un ensemble de modèles de faute M si :*

$$\forall (m, m') \in M^2, m \neq m', S_I^m \cap S_I^{m'} = \emptyset$$

En d'autres termes, un test de caractérisation de faute, vis-à-vis d'un ensemble de modèles de faute M, doit maximiser le nombre d'états finaux fautés (i.e. des états finaux différents de l'état attendu) et les états finaux fautés doivent être distinguables entre les modèles de faute.

Nous pouvons dériver de ces propriétés des métriques de performance pour évaluer par simulation d'injection de faute les tests, à savoir le taux de propagation, ou *propagation rate* (PR), le taux discrimination, ou *discrimination rate* (DR) et le taux de couverture, ou *coverage rate* (CR).

Definition 3 (Taux de propagation) *Le taux de propagation d'un test de caractérisation de fautes, selon l'ensemble des modèles M et la séquence I, se définit comme :*

$$PR_{M,I} = \frac{\#\{\textit{Résultats fautés}\}}{\#\{\textit{Fautes injectées}\}}$$

Definition 4 (Taux de discrimination) *Le taux de discrimination d'un test de caractérisation de fautes, selon l'ensemble des modèles M et la séquence I, se définit comme :*

$$DR_{M,I} = 1 - \frac{\#\{\textit{Résultats fautés communs à plusieurs modèles}\}}{\#\{\textit{Résultats fautés}\}}$$

Definition 5 (Taux de couverture) *Le taux de couverture d'un test de caractérisation de fautes, selon l'ensemble des modèles M et la séquence I, se définit comme :*

$$CR_{M,I} = \frac{\#\{\textit{Modèles couverts}\}}{\#\{\textit{Modèles}\}}$$

Comme nous voulons que les tests de caractérisation propagent, discriminent, couvrent un maximum de fautes, PR, DR et CR doivent être maximisés. Nous pouvons rapidement comparer la performance de deux tests de caractérisation de fautes différents avec le produit de PR, DR et CR. Dans ce qui suit, nous expliquerons comment les choix de conception des tests de caractérisation de fautes influencent PR, DR et CR, en se basant sur une étude des tests les plus fréquemment utilisés.

Catégorie	Famille de modèles de faute	Description
Corruption d'Instruction	F1	Saut d'instructions
	F2	Bit-sets sur l'instruction décodée
	F3	Bit-resets sur l'instruction décodée
Corruption de Registre	F4	Bit-sets sur la valeur du registre lue
	F5	Bit-resets sur la valeur du registre lue
Corruption de mémoire	F6	Bit-sets sur la valeur de la cellule mémoire lue
	F7	Bit-resets sur la valeur de la cellule mémoire lue

TABLE 5.1 – Modèles de fautes considérés.

5.4 Critères de conception

Dans cette section nous évaluons, par simulation d'injection de fautes avec `CELTIC`, la performance de différents tests populaires avec les métriques PR, DR et CR. Ensuite, nous comparons l'influence des différents choix de conception sur les métriques retenues et nous déduisons les critères généraux pour concevoir des tests de caractérisation de fautes selon les modèles de faute supposés.

5.4.1 Les modèles de faute

Nous avons considéré un ensemble classique de modèles de faute au niveau ISA. Cet ensemble, présenté au tableau 5.1, couvre un large éventail de modèles de faute pouvant se produire sur le microcontrôleur cible, de la corruption d'instruction (F1-F3), en passant par la corruption de registre (F4, F5) à la corruption de mémoire (F6, F7).

Les modèles `16InstructionSkip` (saut de 16 octets par rapport à l'adresse normalement chargée), `SetOpcodeBit6` (Bit-set du bit 6 de l'opcode) et `ResetOpcodeBit4` (Bit-reset du bit 4 de l'opcode) sont des exemples de modèles de fautes appartenant à la catégorie de corruption d'instruction. Les modèles `SetRegisterBit12` (Bit-set du bit 12 de la valeur du registre lue) et `ResetRegisterBit8` (Bit-reset du bit 8 de la valeur du registre lue) sont des exemples de modèles de fautes appartenant à la catégorie de corruption de registre. Les modèles `SetMemoryCellBit3` (Bit-set du bit 3 de la valeur de la mémoire lue) et `ResetMemoryCellBit1` (Bit-reset du bit 1 de la valeur de la mémoire lue) sont des exemples de modèles de fautes appartenant à la catégorie de corruption de mémoire.

Ces modèles de faute sont implémentés dans `CELTIC` et seront utilisés pour comparer PR, DR et CR des différents tests de caractérisation de fautes.

5.4.2 Vue générale des tests de caractérisation de fautes

Nous avons sélectionné 8 tests de caractérisation de fautes de la littérature, résumé au tableau 5.2. Ces tests sont présentés en annexe. On peut y retrouver la séquence d'instructions

I et les valeurs initiales V de chaque test. Ces tests de caractérisation de fautes ont été conçus pour comprendre les effets de fautes sur différents microcontrôleurs. Lorsque nécessaire, ces tests ont été traduits vers le jeu d'instructions ARMv7-M 16-bit de manière à comparer les performances plus facilement. Nous avons classifié ces tests selon différents critères :

- L'idempotence des instructions dans I : nous distinguons les instructions idempotentes donnant le même résultat appliquées une ou plusieurs fois de celles non-idempotentes.
- Le nombre d'instructions dans I : les tests sont classifiés selon la taille de la séquence I.
- La variété des instructions dans I : on distingue les tests utilisant plusieurs instructions différentes de ceux utilisant une unique instruction.
- Le type d'instruction dans I : on différencie les instructions logiques et arithmétiques des instructions mémoire.
- Le choix des valeurs initiales de V : Les valeurs initiales des tests peuvent être identiques ou bien distinctes.

Dans la partie suivante, nous mettons en évidence l'influence de ces critères sur les métriques de performance PR, DR et CR. Ces résultats nous serviront par la suite à concevoir de meilleurs tests de caractérisation de fautes.

5.4.3 Comparaison des performances

Les tests sélectionnés ont été évalués avec `CELTIC` par simulation d'injection de fautes. Nous présentons ici le comparatif des 8 tests en fonction des critères définis précédemment. Les métriques de performance PR, CR, DR et $PR * CR * DR$ en fonction des modèles de fautes et des tests de caractérisation de fautes considérés sont présentées en annexe. Ces résultats serviront de référence pour évaluer nos tests à la section 5.5. Avant de rentrer dans les détails, quelques observations générales tirées des résultats obtenus :

- Les modèles de saut d'instruction(s) (F1) ont les résultats avec l'écart type le plus élevé, ce qui suggère que pour ces modèles, PR, DR et CR sont très dépendants des choix de conception.
- Mis à part pour les modèles de corruption de la mémoire, le test T1 est le test le plus générique car il fonctionne très bien avec les modèles de corruption d'instruction (F1,F2,F3) et avec les modèles de corruption de registre (F4,F5).
- Le test T6 est le moins performant pour les modèles testés.

En croisant les résultats des simulations d'injections de faute en fonction des caractéristiques des tests sélectionnés (tableau 5.2), nous pouvons identifier les choix de conception qui influencent le plus PR, DR et CR.

Test	Idempotence	Nombre d'instructions	Type d'instruction
T1	Non	Élevé	Instructions arithmétiques et logiques
T2	Oui	Élevé	Instructions arithmétiques et logiques
T3	Non	Faible	Instructions arithmétiques et logiques
T4	Non	Élevé	Instructions arithmétiques et logiques
T5	Non	Élevé	Instructions arithmétiques et logiques
T6	Non	Faible	Instructions Load and Store
T7	Mixte	Élevé	Instructions arithmétiques et logiques
T8	Oui	Élevé	Instructions Load and Store

Test	Variété des instructions	Valeurs initiales de V	Travaux Récents
T1	Moyenne	Distinctes	[PHM+19]
T2	Moyenne	Distinctes	[TBC19, CMD+18, MDH+13]
T3	Moyenne	Distinctes	[RNR+15, KH14]
T4	Moyenne	Identiques	-
T5	Faible	Distinctes	[PHM+19, TSW16, DRPR19]
T6	Moyenne	Distinctes	[DRPR19, TSW16, BGV11]
T7	Élevée	Distinctes	[BGV11]
T8	Moyenne	Distinctes	[TBC19, DPdC+15]

TABLE 5.2 – Présentation des 8 tests de caractérisation de fautes utilisés dans des travaux récents en fonction de plusieurs critères.

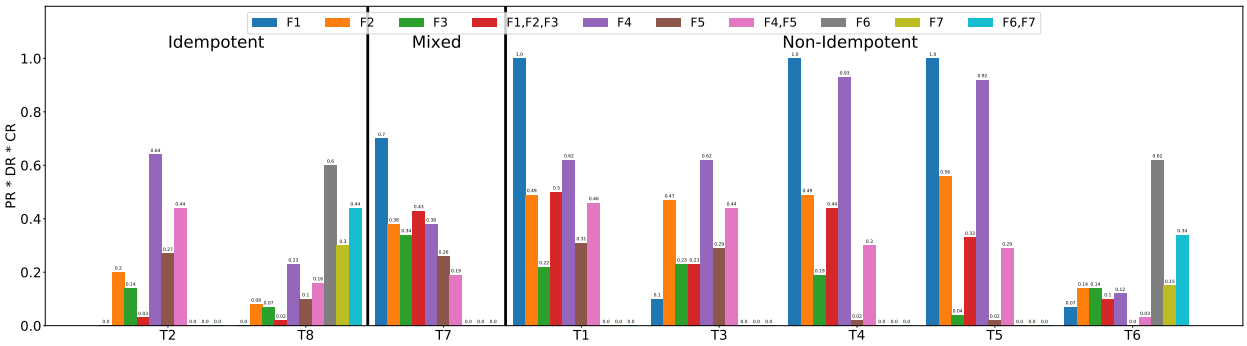


FIGURE 5.1 – Comparaison de $PR * DR * CR$ des tests idempotents avec les tests non-idempotents.

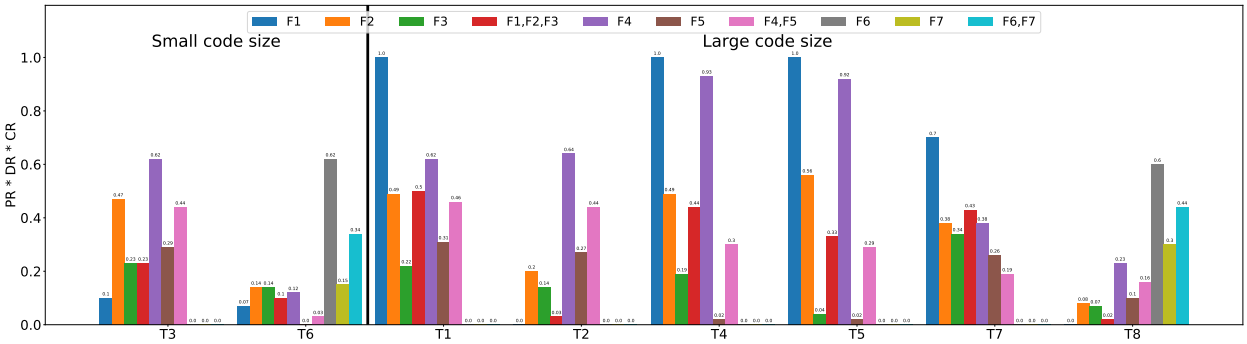


FIGURE 5.2 – Comparaison de $PR * DR * CR$ en fonction du nombre d'instruction.

Idempotence Le fait d'utiliser des instructions ne modifiant pas l'état courant du programme est souvent encouragé pour observer et comprendre les effets des fautes [TBC19, CMD+18] (e.g. test T2). Comme les instructions de la séquence I sont idempotentes pour ces tests, les valeurs finales V attendues sont identiques aux valeurs initiales. Parmi les avantages, le post-traitement des résultats est facilité car l'effet de la faute peut se comprendre manuellement, sans outil de simulation ; car l'effet est observable (s'il se propage) directement en comparant les valeurs V finales et initiales. Cependant, d'après les résultats de la figure 5.1, les tests idempotents (T2 et T8) sont moins performants que leurs homologues non-idempotents. Tout particulièrement pour les modèles de saut d'instruction(s) (F1), les tests idempotents ne propagent pas une seule faute. La seule situation où les tests idempotents peuvent être utilisés est lorsque les modèles de faute supposés affectent seulement les données, registre ou mémoire (F4-F7). *Par conséquent, les tests idempotents sont fortement déconseillés pour les modèles de fautes de corruption d'instruction (F1, F2 et F3).*

Nombre d'instructions Le nombre d'instructions dans la séquence I des tests est un autre facteur important à prendre en compte. Les tests T3 et T6 contiennent entre 8 et 7 instructions dans I tandis que les autres tests évalués contiennent autour de 80 instructions. D'après les résultats de la figure 5.2, les taux de propagation et de couverture sont plus élevés pour les tests avec un nombre d'instructions élevés, en particulier pour les modèles de faute de saut d'instruction(s) (F1). De plus, expérimentalement, il est plus difficile de cibler la séquence I avec un nombre d'instructions faible. Dans ce cas, si un état final fauté est observé, il est facile de se tromper sur l'effet de la faute car la faute a pu se produire en dehors de la séquence I. Par conséquent, il n'y pas de raison particulière à utiliser un nombre d'instructions faible, et *nous conseillons fortement d'utiliser un nombre d'instructions élevé.*

Variété d'instructions Si la majorité des tests évalués contiennent entre 7 et 8 instructions différentes dans la séquence I (variété moyenne), le test T5 ne contient qu'une seule instruction (variété faible) tandis que le test T7 utilise 80 instructions différentes (variété élevée). En regardant les résultats du taux de couverture (figure A.VI en annexe), en particulier les modèles de faute de corruption instruction (F1-F3) et de corruption de registre (F4, F5), Le test T7 est meilleur que le reste du benchmark. Au contraire, le taux de discrimination

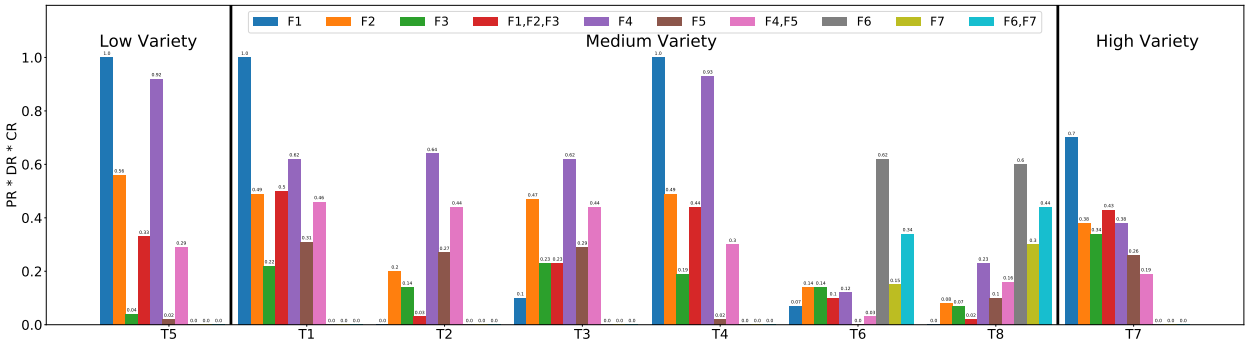


FIGURE 5.3 – Comparaison de $PR * DR * CR$ en fonction de la variété des instructions.

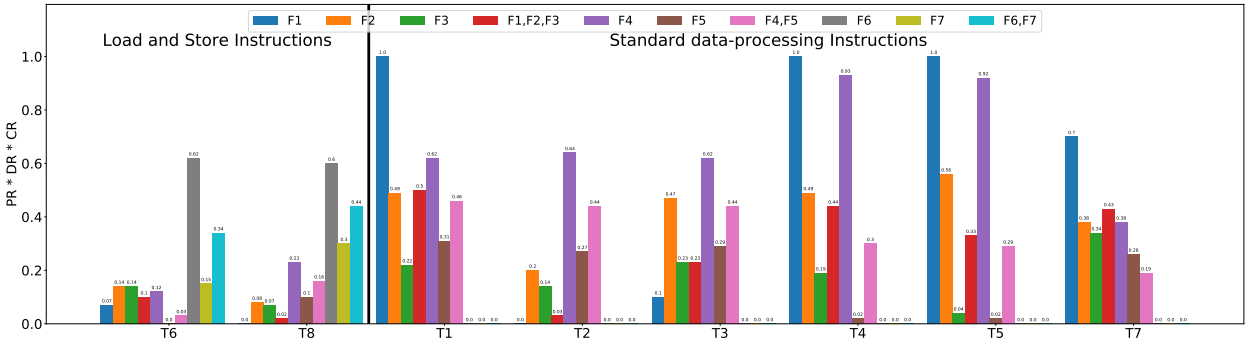


FIGURE 5.4 – Comparaison de $PR * DR * CR$ en fonction du type d'instruction.

(figure A.V en annexe) pour ces mêmes modèles est significativement inférieur en utilisant le test T7. Cela suggère qu'utiliser une variété d'instructions élevée tend à augmenter le taux de couverture de fautes au détriment du taux de discrimination. À l'inverse, utiliser une variété d'instructions faible, comme pour le test T5, permet de propager, discriminer et couvrir une famille de modèles de faute en particulier et pas les autres (figure 5.3). Par exemple, le test T5 est bien adapté pour observer les bit-set sur l'instruction (F2) mais pas les bit-reset (F3). Si les résultats expérimentaux ne permettent pas de trancher pour une variété faible ou élevée, à la section 5.5, *nous proposons des tests optimaux avec une variété d'instruction faible pour chaque modèle de faute du tableau 5.1.*

Type d'instruction Sans surprise, seuls les tests utilisant des instructions *load* et *store* (T6 et T8) peuvent propager les modèles de faute de corruption mémoire (F6, F7). Plus intéressant, on peut observer sur la figure 5.4 que les instructions arithmétiques et logiques sont meilleures que les instructions *load* et *store* pour propager les modèles de fautes de corruption instruction et corruption registre (F1-F5). *Par conséquent, mis à part pour observer les effets de fautes sur la mémoire, les instructions arithmétiques et logiques sont conseillées.*

Valeurs initiales En fonction des modèles de faute considérés, les valeurs initiales de V peuvent influencer PR et CR. Parmi les tests sélectionnés, seul le test T4 initialise les variables V à zéro, tandis que les autres tests initialisent les variables V avec des valeurs

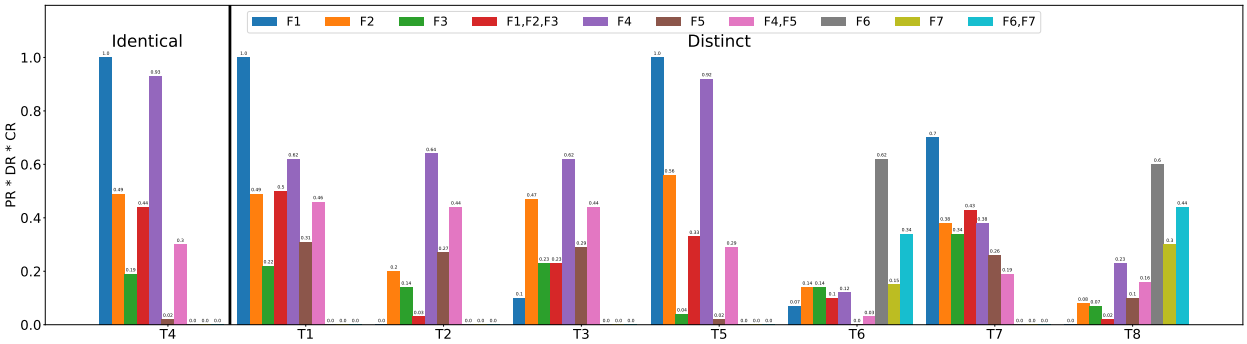


FIGURE 5.5 – Comparaison de $PR * DR * CR$ en fonction des valeurs initiales.

distinctes. Les valeurs initiales de V ne sont pas impactantes dans le cas des modèles de corruption d'instruction (F1-F3). À l'inverse, en se basant sur les résultats de la figure 5.5, les valeurs initiales influencent les taux de propagation des modèles de fautes de corruption de registre (F4, F5). Similairement, modifier les valeurs initiales chargées en mémoire des tests T6 et T8 influencent la propagation de modèles de corruption de mémoire (F6, F7). *En conséquence, nous recommandons de choisir avec soin les valeurs initiales de V pour les familles de modèles de faute sur les données, registre ou mémoire (F4-F7).*

5.4.4 Critères de conception proposés

Pour résumer, le tableau 5.3 récapitule les critères de conception les plus importants en fonction des modèles de fautes du tableau 5.1. Certains critères peuvent sembler vagues ; et certaines questions subsistent, par exemple, *quelles instructions arithmétiques et logiques doit on choisir pour I ?* Ou bien encore *comment initialiser les variables V ?*

Il est malheureusement impossible de répondre de manière générale à ces questions, car les instructions ou les valeurs précises dépendent bien évidemment des modèles de fautes, mais aussi de l'architecture du microcontrôleur ciblé.

Néanmoins, dans la prochaine section, nous détaillons précisément comment choisir les valeurs initiales de V et les instructions optimales de I, en s'appuyant sur les critères de conception que nous venons de mettre en évidence, afin de concevoir des tests de caractérisation de fautes optimaux pour chacun des modèles de fautes considérés.

5.5 Concevoir des tests optimaux

Les tests de caractérisation de fautes des travaux précédents sont sous-optimaux, pour la majorité des modèles de fautes évalués avec *CELTIC* (à l'exception du modèle de saut d'instructions F1), dans le sens où $PR * DR * CR < 1$.

Dans cette section, nous concevons sept tests optimaux maximisant PR, DR et CR pour chaque modèle de faute. Cependant, comme les évaluations de sécurité se déroulent souvent en temps contraint, il n'est généralement pas possible d'utiliser un grand nombre de tests différents pour calibrer l'équipement et/ou caractériser le composant. Dans le but

Catégorie	Famille	Critères Importants
Corruption d'Instruction	F1	Instructions arithmétiques et logiques Non-idempotence Nombre d'instructions élevé
	F2,F3	Instructions arithmétiques et logiques Non-idempotence
Corruption de Registre	F4,F5	Instruction arithmétiques et logiques Idempotence Valeurs Initiales de V
Corruption de mémoire	F6,F7	Instructions Load et Store Idempotence Valeurs Initiales de V

TABLE 5.3 – Critères de conception les plus importants en fonction des modèles du tableau 5.1.

I		V	
`adds r0, #1` `adds r0, #1`	} Repeat n times	`R0`	`0x00000000`

TABLE 5.4 – Test de caractérisation optimal pour les modèles de saut d'instructions (F1).

de réduire le temps consacrer à la caractérisation, nous proposons un ensemble minimal de trois tests permettant de propager, discriminer, et couvrir l'ensemble des modèles de fautes du tableau 5.1.

5.5.1 Tests optimaux proposés

Un test est dit optimal s'il obtient $PR * DR * CR = 1$. Pour trouver les tests optimaux pour chaque modèle, nous allons nous appuyer sur les observations de la section 5.4, en particulier, sur le fait qu'une variété faible de la séquence I tend à spécialiser le test pour une famille de modèles de faute. Ainsi, l'objectif est de trouver l'unique instruction i répétée n fois, ainsi que les valeurs initiales V maximisant PR, DR et CR, pour chaque modèle du tableau 5.1.

5.5.1.1 Saut d'instructions (F1)

Test proposé Ce test est similaire aux travaux précédents ([PHM+19, TSW16, DRPR19]). La séquence I est une addition avec une valeur immédiate répétée n fois, avec n plus grand que le plus grand saut possible supposé (tableau 5.4). Les valeurs initiales de V n'ont aucune influence et peuvent être choisies arbitrairement. Ce test est optimal ($PR * DR * CR = 1$) pour les modèles de saut d'instructions.

I		V	
`mov r0, r0` `mov r0, r0`	} Repeat n times	`R0`	`0x00000000`

TABLE 5.5 – Test de caractérisation optimal pour les modèles de bit-set sur la valeur du registre source (F4).

I		V	
`mov r0, r0` `mov r0, r0`	} Repeat n times	`R0`	`0xffffffff`

TABLE 5.6 – Test de caractérisation optimal pour les modèles de bit-reset sur la valeur du registre source (F5).

Discussion L'instruction i doit mettre à jour l'état courant vers un nouvel état distinct, et ceci pour chaque exécution de i. Le nombre d'états distincts doit être supérieur au nombre supposé d'instruction sautées. Par exemple, une *addition avec une valeur immédiate* permet d'obtenir facilement n états distincts en incrémentant à chaque exécution le registre de destination. Au contraire, un *ou exclusif* ne permet d'obtenir que 2 états distincts et ne sera pas adapté pour les modèles de saut d'instructions.

5.5.1.2 Corruption de registre (F4, F5)

Test proposé Ce test est similaire aux travaux existants ([CMD+18, TBC19]). La séquence I est une instruction *mov* entre le même registre (tableau 5.5, tableau 5.6). Le nombre d'instructions n peut être choisi arbitrairement. La valeur du registre utilisé doit être initialisée à `0x0...0` (`0xf...f`) pour les bit-set (bit-reset). Ces tests sont optimaux ($PR*DR*CR = 1$) pour les modèles de bit-set (bit-reset) sur la valeur du registre source.

Discussion L'instruction i doit être idempotente pour ne pas mettre à jour l'état courant. Ainsi, si un bit-set (bit-reset) se produit sur la valeur du registre, la valeur corrompue sera propagée, et l'état final fauté dépendra seulement de la valeur du bit corrompu. L'instruction *mov* entre le même registre remplit parfaitement ce rôle.

5.5.1.3 Corruption de mémoire (F6, F7)

Test proposé Ce test est similaire aux travaux existants ([TBC19]). La séquence I est composée d'une instruction *store* suivie immédiatement d'une instruction *load* permettant de lire et écrire la même valeur à la même adresse mémoire (tableau 5.7, tableau 5.8). Le nombre d'instructions n peut être choisi arbitrairement. Le registre tampon et la valeur en mémoire V doivent être initialisées à `0x0...0` (`0xf...f`) pour les bit-set (bit-reset). Ces tests sont optimaux ($PR*DR*CR = 1$) pour les modèles de bit-set (bit-reset) sur la valeur de la cellule mémoire.

I		V	
`ldr r0, =#0x20002000 ; [r0] = 0x00000000`		`R0`	`0x20002000`
`str r1, [r0]`	} Repeat n times	`R1`	`0x00000000`
`ldr r1, [r0]`		`[R0]`	`0x00000000`

TABLE 5.7 – Test de caractérisation optimal pour les modèles de bit-set sur la valeur de la cellule mémoire (F6).

I		V	
`ldr r0, =#0x20002000 ; [r0] = 0xffffffff`		`R0`	`0x20002000`
`str r1, [r0]`	} Repeat n times	`R1`	`0xffffffff`
`ldr r1, [r0]`		`[R0]`	`0xffffffff`

TABLE 5.8 – Test de caractérisation optimal pour les modèles de bit-reset sur la valeur de la cellule mémoire (F7).

Discussion Comme pour les corruptions de la valeur du registre, l'instruction i doit être idempotente pour ne pas mettre à jour l'état courant. Cependant, il n'existe pas d'instruction atomique permettant de lire et écrire en mémoire, et donc, nous devons utiliser exceptionnellement deux instructions.

5.5.1.4 Corruption d'opcode (F2, F3)

Vue Générale Contrairement aux tests précédents, les travaux récents ne proposent pas de solutions optimales pour propager, discriminer et couvrir les effets de bit-set et bit-reset sur l'opcode chargé. Nous proposons d'utiliser `CELTIC` (Algorithme 2) pour trouver systématiquement l'instruction optimale, i en fonction du jeu d'instruction cible. Le nombre d'instructions n et les valeurs initiales de V peuvent être choisis arbitrairement.

Algorithme Trouver l'instruction optimale pour propager des corruptions de l'opcode est plus difficile que pour les autres familles de modèles de faute. La raison principale vient du fait que l'instruction optimale dépend du jeu d'instruction. Les opcodes fautés doivent être valides selon le jeu d'instruction, tout en veillant à ne pas masquer la faute.

Pour faciliter la recherche de l'opcode optimal pour propager les effets de bit-set (bit-reset) sur l'opcode, nous proposons l'Algorithme 2. L'idée principale derrière cet algorithme est que le PR maximal possible pour un opcode à une distance de *hamming* d de l'opcode initial `0x0...0` (`0xf...f`) est $1 - d/size$, avec $size$ la taille d'instruction en bit.

En prenant soin de trier les opcodes par nombre de bits déjà set (reset), le premier opcode dont le PR associé est supérieur ou égal à $1-d/size$ est l'opcode du jeu d'instruction propageant le plus de faute selon le modèle F2 (F3). Egalement, DR ne dépend pas de la taille d'instruction ou de la distance de *hamming* et $CR = PR$ car la séquence d'instructions

Algorithme 2 : Recherche de l'opcode optimal pour propager les effets de bit-set (bit-reset) sur l'opcode.

Input : Jeu d'instructions J, taille d'instruction en bit $size$
Output : Opcode optimal op_{best}.

```
/* Get encodings w.r.t J sorted by number of bits already set (reset) */
E ← GetEncodingsSorted(J);
op_best ← ∅ ; max ← 0 ; d ← 0;
/* While max is below the maximum PR at hamming distance d */
while max < 1 − d/size do
    j ← 0;
    /* While number of bits already set (reset) are less or equal to d */
    while Count(E[j]) <= d do
        /* Get operand bits of encoding E[j] */
        B ← GetOperandBits(E[j]);
        /* Init op_init of encoding E[j] with all operand bits reset (set) */
        op_init ← InitAllOperandBits(E[j]);
        /* Generate all bit-sets (bit-resets) on B, up to d, minus bits already set
           (reset) */
        C ← Combinations(B, d−Count(E[j]));
        foreach c ∈ C do
            /* Init op_test with operand bits set (reset) w.r.t c */
            op_test ← GenOperandBits(op_init, c);
            /* Repeat opcode 2 times, prevent idempotent solutions */
            test ← [op_test] ∗ 2;
            stats ← SimulateFI(test, m);
            PR, DR ← GetRates(stats);
            if PR ∗ DR > max then
                max ← PR ∗ DR;
                op_best ← op_test;
        j ← j + 1;
    d ← d + 1;
return op_best;
```

I ne contient qu'une seule instruction répétée n fois. Ainsi, l'opcode optimal pour le jeu d'instruction J est le premier dont $PR * DR$ est supérieur ou égal à $1 - d/size$.

Par exemple, on souhaite trouver la meilleure instruction 16-bit pour observer des bit-sets (single-bit) sur instruction (famille de modèles F2). Dans ce cas, l'instruction permettant d'observer théoriquement tous les bit-sets est `0x0000`. Le taux de propagation maximal possible est 1. Si le $PR = 1$ pour `0x0000`, `0x0000` est l'instruction optimale.

Mais, en pratique, il est possible que l'instruction fautée obtenue n'existe pas dans l'ISA ou bien que l'instruction fautée obtenue n'engendre pas un résultat fauté observable ; et que

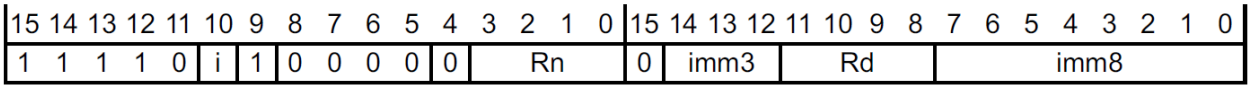


FIGURE 5.6 – Encodage de l'instruction ARMv7-M `ADDW<c> <Rd>, <Rn>, #<imm12>`.

Modèle	Instruction ARMv7-M 16-bit Optimale			
	Opcode	Mnémonique	PR	DR
F2	`3201`	`adds r2, #1`	0.75	1.00
F3	`3fff`	`subs r7, #ff`	0.875	1.00

TABLE 5.9 – Instructions optimales 16-bit du jeu d'instructions ARMv7-M trouvées avec l'Algorithme 2.

par conséquent le $PR < 1$ pour l'instruction `0x0000`.

Si $PR < 1$ pour `0x0000`, le PR maximal devient $1 - d/size = 0.975$ car on doit considérer à présent toutes les instructions avec une distance de hamming d de 1 (toutes ces instructions ont un bit déjà set, limitant la propagation). Ainsi, si $PR >= 0.975$ pour l'instruction `0x0000`, `0x0000` reste l'instruction optimale. Sinon on calcule le PR des instructions `0x1000`, `0x2000`, `0x4000`, etc. (soit toutes les instructions $d = 1$). Puis les instruction $d = 2$, etc. jusqu'à ce qu'une instruction obtienne un $PR >= 1 - d/size$.

De plus, il faut tenir compte du nombre d'instructions, qui sans optimisation, est rapidement problématique. Par exemple, nous trouvons que l'opcode optimal pour l'architecture ARMv7-M 32-bit pour propager un effet de bit-set sur l'opcode est à une distance de *hamming* de 7 de l'opcode initial `0x0...0` (tableau 5.10). Sans optimisation, il faut tester toutes les combinaisons à une distance de *hamming* de 7, ce qui nous donnes $\sum_{k=0}^{7} \binom{32}{k} \approx 10^6$ combinaisons possibles à simuler avec `CELTIC`.

De manière à réduire l'explosion combinatoire, en particulier pour les instructions 32-bit ou plus, nous utilisons les motifs d'encodage implémentés dans *CELTIC* [Dur16]. Au lieu de tester tous les opcodes jusqu'à une distance de *hamming* de d, seuls les bits des opérandes B des encodages valides sont utilisés. Par exemple, seuls les bits des opérandes de l'encodage `i`, `Rn`, `imm3`, `Rd` et `imm8` de la figure 5.6 sont considérés pour générer les possibles bit-set (bit-reset) jusqu'à une distance de *hamming* d de l'opcode initial, moins le nombre de bits déjà *set* (*reset*) de cet encodage.

Modèle	Instruction ARMv7-M 32-bit Optimale			
	Opcode	Mnémonique	PR	DR
F2	`f1400001`	`adc r0, r0, #1`	0.67	0.90
F3	`f57777ff`	`sbcs r7, r7, #0x1fe`	0.70	0.94

TABLE 5.10 – Instructions optimales 32-bit du jeu d'instruction ARMv7-M trouvées avec l' Algorithme 2.

I		*V*	
`subs r7, #ff` `subs r7, #ff`	} Repeat n times	`R7`	`0x00000000`

TABLE 5.11 – Test de caractérisation optimal pour les modèles de bit-reset sur l'opcode (F3).

Exemple Nous avons sélectionné comme jeu d'instructions J un sous-ensemble du jeu d'instruction de l'architecture ARMv7-M en 16-bit et 32-bit (tableau A.V). À partir de ce jeu d'instruction, nous pouvons utiliser 54 encodages 16-bit et 222 encodages 32-bits différents. Les opcodes optimaux 16-bit et 32-bit pour propager les effets des familles de modèles de fautes F2 et F3 sont présentés au tableau 5.9 et tableau 5.10. Le tableau 5.11 présente un exemple de test de caractérisation optimal pour les modèles de bit-reset sur l'opcode (F3), basé sur l'instruction optimale `subs r7, #ff` trouvée avec Algorithme 2.

Pour référence, les meilleurs tests évalués à la section 5.4 ont un $PR * DR * CR = 0.56$ pour la famille de modèles F2 (*test T5*) et 0.34 pour la famille de modèles F3 (*test T1*). En comparaison, nos tests avec les instructions 16-bit du tableau 5.9 atteignent 0.56 pour F2 et 0.77 pour F3. Si, pour la famille de modèles F2, le *test T5* est déjà optimal, pour la famille de modèles F3, notre test (tableau 5.11) est deux fois plus performant que le *test T1* d'après nos métriques.

5.5.2 Ensemble minimal de tests

Les tests optimaux présentés sont spécialisés pour une famille de modèles de faute. Ainsi pour couvrir l'ensemble des familles du tableau 5.1, il faut recourir à 6 tests différents (le test F1 étant équivalent au test F2). Lors d'évaluations de sécurité en temps contraint, il n'est possible de faire une caractérisation de fautes avec 6 tests différents car extrêmement chronophage. Pour cette raison, nous proposons un ensemble minimal de tests permettant de couvrir l'ensemble des modèles du tableau 5.1. Les tests optimaux présentés à la sous-section précédente sont regroupés pour former trois tests couvrant les trois catégories de familles de modèles de faute considérées ; corruption d'instruction (F1-F3), corruption de registre (F4,F5) et corruption de mémoire (F6,F7). Ces tests sont les suivants :

- Le test de corruption d'instruction (IC) (tableau 5.12) contient une alternance des instructions optimales trouvées avec l'Algorithme 2, permettant de propager, discriminer et couvrir les effets de saut d'instructions mais aussi les effets de bit-set et bit-reset sur l'opcode.
- Le test de corruption de registre (RC) (tableau 5.12) est basé sur les tests optimaux pour les familles de modèles F4 et F5, propageant à la fois les effets de bit-set et bit-reset sur les registres (tableau 5.5,tableau 5.6).
- Le test de corruption de mémoire (MC) (tableau 5.13) regroupe les tests optimaux pour les familles de modèles F6 et F7, propageant les effets de bit-set et bit-reset sur

	Test de corruption d'instruction (IC)				Test de corruption de registre (RC)			

I	`adds r2, #1`
	`subs r7, #ff`
	`adds r2, #1`
	`subs r7, #ff` } Repeat n times
					`mov r0, r0`
					`mov r1, r1`
					`mov r0, r0`
					`mov r1, r1` } Repeat n times			

V	R0	0x00000000	R1	0x11111111	R0	0x00000000	R1	0xffffffff
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE 5.12 – Test de corruption d'instruction (IC) et Test de corruption de registre (RC) pour le jeu d'instruction ARMv7-M.

	Test de corruption de mémoire (MC)			

I	`ldr r0, =#0x20002000 ; [r0] = 0x00000000`
	`ldr r2, =#0x20002004 ; [r2] = 0xffffffff`
	`str r1, [r0]`
	`ldr r1, [r0]`
	`str r3, [r2]`
	`ldr r3, [r2]` } Repeat n times			

V	R0	0x20002000	R1	0x00000000
	R2	0x20002004	R3	0xffffffff
	[R0]	0x00000000	[R2]	0xffffffff

TABLE 5.13 – Test de corruption de mémoire (MC) pour le jeu d'instruction ARMv7-M.

la mémoire (tableau 5.7,tableau 5.8).

La performance de ces tests a été évaluée par simulation d'injections de faute en utilisant `CELTIC`. Le tableau 5.14 détaille les métriques PR, DR, CR pour les tests IC, RC et MC. Selon nos métriques de performance, les tests proposés propagent, discriminent et couvrent environ 8% plus de fautes que les meilleurs tests des travaux précédents.

Nous avons également essayé de combiner les tests optimaux proposés ci-dessus en un seul test de caractérisation de fautes, mais cela fait chuter considérablement PR, DR et CR. Cela est dû au fait que les instructions des tests optimaux sont spécialisées pour une seule famille de modèles. Par exemple, en combinant des instructions non-idempotentes et idempotentes dans le même test, cela implique nécessairement un compromis entre la propagation des effets de faute des familles de modèles F1 (saut d'instructions) et F4-F7 (corruption de données).

5.6 Cas pratique d'utilisation

Pour mettre la théorie en pratique, nous allons utiliser notre ensemble minimal de tests expérimentalement avec notre `GLITCH STATION` sur un microcontrôleur 32-bit. Avec ces tests,

Test	Famille	PR	DR	CR	$PR * DR * CR$	Référence $PR * DR * CR$
Corruption d'instruction	F1,F2,F3	0.66	0.93	0.88	**0.54**	0.50 (T1)
Corruption de registre	F4,F5	0.50	1.00	1.00	**0.50**	0.46 (T1)
Corruption de mémoire	F6,F7	0.50	1.00	1.00	**0.50**	0.44 (T8)

TABLE 5.14 – PR, DR, et CR pour le test IC, RC et MC et comparaison avec le meilleur test des travaux existants.

nous pouvons identifier la catégorie de modèles de faute la plus probable sur la cible. Également, en fonction du test utilisé, nous verrons que nous pouvons obtenir des paramètres d'équipement générant des fautes différentes.

5.6.1 Protocole expérimental

Cible utilisée Les tests de caractérisation de fautes IC, RC et MC ont été évalués dans les mêmes conditions sur la même cible, à savoir un microcontrôleur 32-bit. Ce dernier embarque un cœur ARM Cortex-M4 cadencé à 72Mhz, reposant sur une architecture Harvard ARMv7E-M avec un *pipeline* de trois étages (chargement, décodage, exécution) et prédiction de branchement.

Protocole Nous avons utilisé notre `GLITCH STATION` pour évaluer la cible à l'aide des tests IC, RC et MC. La technique d'optimisation des paramètres d'équipement retenue est basée sur un algorithme génétique (chapitre 2). En particulier, nous avons injecté 150.000 fautes, sur une période d'environ 12 heures, pour évaluer chacun des tests IC, RC et MC.

5.6.2 Résultats

Les résultats des expérimentations sont détaillés à la figure 5.7, montrant l'évolution de la probabilité de faute sur les 100 premières générations de paramètres d'équipement pour chaque test (soit 25.000 fautes). Également, la figure 5.8 présente les formes de glitch avec la probabilité de faute la plus élevée, obtenues en utilisant chacun des tests de caractérisation de fautes IC, RC et MC. Les principaux points à retenir des expériences sont discutés dans ce qui suit :

Validation de modèles de faute En analysant l'évolution de la probabilité de faute (figure 5.7) des différents tests de caractérisation de fautes, nous validons (ou réfutons) des hypothèses de modèles de faute. D'après les taux de propagation, de discrimination, et de couverture en fonction des différentes familles de modèles de faute avec les tests IC, RC et MC (tableau 5.14), on déduit des résultats de l'expérimentation qu'il est très peu probable

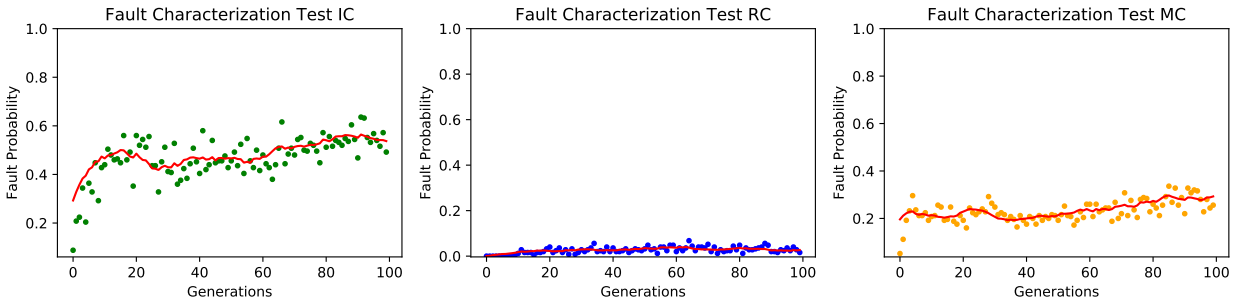


FIGURE 5.7 – Évolution de la probabilité de faute sur les 100 premières générations (25.000 fautes) avec les tests IC, RC et MC, en utilisant notre `GLITCH STATION`. Les lignes rouges sont des moyennes glissantes sur 10 générations.

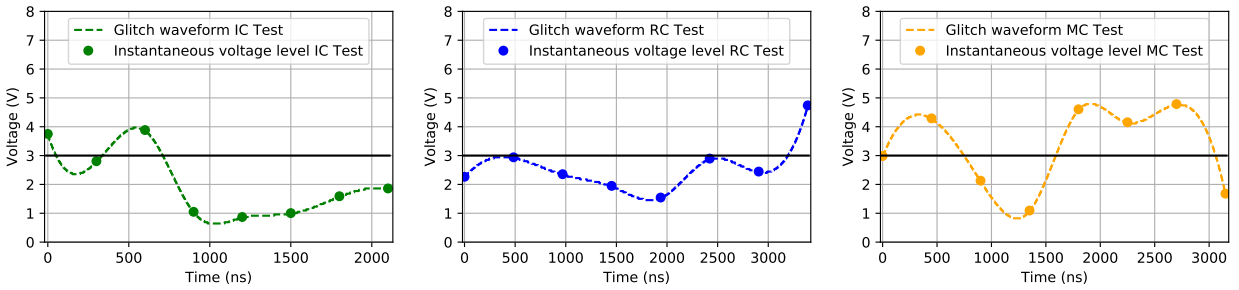


FIGURE 5.8 – Formes de glitch avec la probabilité de faute la plus élevée obtenues avec les tests IC, RC et MC, après 600 générations (150.000 fautes).

d'induire des effets conduisant à des bit-set/bit-reset sur les registres, car la probabilité de faute avec le test RC est très faible (< 0.1). À l'inverse, il est très probable d'obtenir une corruption de l'instruction exécutée d'après les résultats, car la probabilité de faute avec le test IC est élevée (> 0.5). En analysant les états finaux fautés du test IC, il s'est avéré que plus de 90% des états fautés sont causés par un saut d'une ou plusieurs instructions. Ces informations, obtenues facilement via notre ensemble de tests, peuvent s'avérer par la suite décisives lors d'évaluations de sécurité, car un attaquant pouvant injecter des sauts d'instructions de manière contrôlée peut alors construire des attaques complexes [PCHR20].

Spécialisation des paramètres d'injection de fautes Les formes de glitch obtenues avec les différents tests de caractérisation de fautes sur le même microcontrôleur sont radicalement différentes (figure 5.8). Cela est dû au fait que les tests IC, RC et MC ne propagent pas les mêmes effets. Par exemple, comme le test RC ne propage pas les sauts d'instructions, la forme de glitch trouvée avec RC doit nécessairement provoquer un effet différent d'un saut d'instruction. En pratique, cela permet de constituer un ensemble de paramètres d'injection de fautes optimisés conduisant à des effets différents. Une utilisation directe est l'exploitation de vulnérabilité nécessitant plusieurs fautes avec des effets différents comme détaillé dans le chapitre précédent (chapitre 4).

Facilité d'utilisation et applicabilité Avec chaque test de caractérisation de faute, seulement 100 générations (sur les 600), soit 25.000 fautes, sont nécessaires pour identifier des formes de glitch optimisées (figure 5.7) pour ce microcontrôleur. Par conséquent, cet ensemble de tests peut être envisagé pour caractériser des fautes, même lors d'évaluations de sécurité avec de fortes contraintes temporelles. De plus les résultats, (les paramètres d'injection de faute) sont transférables entre les microcontrôleurs avec la même référence, ce qui permet de rentabiliser le temps initialement investi passé à la caractérisation. Ainsi, les mêmes paramètres d'injection de faute peuvent être utilisés pour évaluer différentes applications sur le même microcontrôleur.

5.7 Conclusion

Dans ce chapitre, nous avons proposé des tests optimaux de caractérisation de fautes pour mieux comprendre les effets des fautes sur les microcontrôleurs. Plus précisément, nous avons évalué l'influence de différents choix lors de la conception de tests de caractérisation de fautes, sur la propagation et la discrimination des effets des fautes.

Après avoir défini ces propriétés, nous avons dérivé trois métriques de performance PR, DR et CR dans le but de comparer des tests de caractérisation de fautes issus de la littérature avec `CELTIC` et mettre en évidence les choix de conception les plus importants en fonction des modèles de fautes supposés. Ensuite, sur la base de ces observations et de `CELTIC`, nous avons trouvé des tests de caractérisation de fautes optimaux qui maximisent PR, DR et CR pour des modèles de fautes spécifiques. Puis nous avons combiné ces tests optimaux pour former un ensemble minimal de trois tests couvrant un large éventail d'effets de faute pouvant survenir en pratique. Par ailleurs, les tests que nous avons générés sont facilement généralisables à n'importe quel jeu d'instructions, bien que cela ne soit pas montré dans cette étude.

De plus, nous avons présenté un cas d'utilisation sur un microcontrôleur Cortex-M4 32-bit avec notre `GLITCH STATION`. Notre ensemble minimal de tests permet de facilement identifier les meilleures formes de glitch en moins de 20.000 injections de faute, mais aussi de valider des hypothèses de modèles de faute directement en comparant les résultats des différents tests. La possibilité de trouver rapidement plusieurs paramètres d'équipement conduisant à des effets différents peut considérablement aider à mener des attaques multi-fautes complexes (chapitre 4) qui dépendent en grande partie de l'étape de caractérisation. Enfin, il serait intéressant d'appliquer nos tests en utilisant d'autres techniques d'injection de fautes, pour étudier des effets de fautes plus inhabituels. Idéalement, pour valider la pertinence de notre ensemble de tests, il faudrait réaliser une étude de grande échelle sur une grande variété de microcontrôleurs avec différentes techniques d'injection, ce qui n'a pas pu être réalisé ici.

Toujours dans la démarche d'améliorer l'étape de caractérisation, nous allons nous intéresser aux nouvelles techniques d'optimisation pour identifier les meilleurs paramètres d'équipement dans le chapitre suivant.

Chapitre 6

Identifier et exploiter des vulnérabilités en boîte noire

Sommaire

6.1 Introduction

Ce chapitre décrit en détail notre deuxième approche permettant d'identifier et exploiter des vulnérabilités à l'injection de fautes, mais cette fois-ci, en boîte noire. L'approche proposée est le résultat d'une étude des techniques d'optimisation récentes dans le but d'améliorer l'étape de caractérisation et combler les limitations évoquées au chapitre 4. Ainsi, ce chapitre met l'accent au niveau de l'optimisation des paramètres d'équipement.

Comme nous l'avons vu au chapitre 2, optimiser les paramètres d'équipement est sans doute l'étape la plus importante, car nécessaire pour obtenir des fautes intéressantes. Chaque équipement, en fonction de la technique d'injection de faute, a plusieurs paramètres spécifiques devant être ajustés précisément, telles que les positions x,y,z de la pointe d'une sonde d'un équipement d'injection électromagnétique. Comme l'espace des paramètres est trop grand pour être entièrement couvert, des méthodes ou des techniques d'optimisation ont été proposées pour réduire le temps passé à optimiser l'équipement.

Ainsi nous proposons d'utiliser pour la première fois, dans le contexte de l'injection de faute, de nouvelles techniques d'optimisation qui ont déjà fait leurs preuves dans d'autres

domaines, comme notamment l'apprentissage automatique. Ces techniques permettent de significativement améliorer l'exploration de l'espace des paramètres d'équipement. L'approche proposée est applicable en boîte noire, c'est-à-dire lorsque l'attaquant n'a pas accès à l'application cible. Ainsi, contrairement à l'approche en boîte grise proposée au chapitre 4, l'application ne sera pas analysée avec `CELTIC`. Pour identifier plus facilement les zones critiques de l'application, les paramètres d'équipement sont optimisés séparément des délais d'injection.

La suite du chapitre s'organise comme suit. Après avoir présenté à la section 6.2 nos motivations et l'état de l'art des méthodes et techniques permettant d'explorer plus efficacement l'espace des paramètres, nous détaillerons notre approche et notamment les techniques d'optimisation retenues. Tout d'abord, à la section 6.3, nous présenterons l'approche générale qui vise à séparer l'optimisation du délai d'injection des paramètres d'équipement en utilisant des tests de caractérisation de fautes et de nouvelles techniques d'optimisations n'ayant pas été encore appliquées dans le domaine de l'injection de faute ; puis à la section section 6.4, nous détaillerons ces dernières. La section 6.5 est un comparatif entre plusieurs techniques d'optimisation, visant à mettre en évidence les forces et faiblesses des techniques proposées. Enfin, à la section 6.6, nous évaluerons notre approche sur un cas d'utilisation réel : contourner des contremesures de protection de code d'un microcontrôleur 32-bit.

Les travaux de ce chapitre ont été acceptés à la conférence CARDIS 2021.

6.2 Motivations

6.2.1 Contexte

Parmi les applications régulièrement prises pour cible dans la littérature, on retrouve en premières lignes les *bootloaders* préchargés de microcontrôleurs. Ces programmes permettent généralement d'avoir un accès complet en lecture et en écriture à la mémoire Flash et RAM du microcontrôleur, ce qui explique pourquoi ils sont régulièrement ciblés par les attaquants (chapitre 2). Pour cette raison, les fabricants rajoutent plusieurs contremesures permettant de restreindre les accès mémoires.

Une grande majorité des attaques publiées sur les *bootloaders* embarqués traitent ces derniers en tant que boîtes noires. Ainsi, il faut réussir à trouver les paramètres d'attaque sans connaissance du binaire. Plusieurs approches ont été proposées pour trouver plus rapidement les paramètres d'attaques dans ces conditions.Nous avons déjà présenté au chapitre 2 les différentes stratégies mises en place pour adresser cette problématique. Pour rappel, ces approches consistent à réduire l'espace des paramètres et/ou utiliser des techniques pour optimiser l'exploration de l'espace des paramètres.

D'une part, pour réduire l'espace des paramètres, la majorité des stratégies proposées sont propres à la technique d'injection de faute utilisée. Pour l'injection de faute par perturbation laser, les zones d'intérêt sur la puce sont mis en évidence avec un microscope à balayage [CLMFT14] ou en mesurant le courant induit par faisceau optique [SFR+15]. Similairement, pour l'injection de faute par perturbation électromagnétique, mesurer préalablement l'émanation électromagnétique de la puce permet de cibler les zones d'intérêt

[MAM17]. Toutefois, une stratégie de réduction de l'espace des paramètres, facilement généralisable à l'ensemble des techniques d'injection de faute, consiste à isoler l'optimisation des paramètres d'équipement des délais d'injection [CPB+13].

D'autre part, pour optimiser l'exploration des paramètres, on peut utiliser la recherche par quadrillage (*Grid Search*, GS) ou la recherche aléatoire (*Random Search*, RS). Ces deux techniques présentent l'avantage d'être facilement utilisables, mais sont limitées par la taille de l'espace des paramètres à explorer (chapitre 2). Récemment, les techniques d'optimisation populaires reposent majoritairement sur des algorithmes métaheuristiques, en particulier des algorithmes génétiques (*Genetic Algorithm*, GA) [CPB+13, PBBJ15, PBJC14, BFP19, MSPB19]. Plus récemment, de l'apprentissage profond a même été proposé pour optimiser la puissance et largeur d'impulsion d'une diode laser [WRBBP20].

Néanmoins, la limitation principale des algorithmes métaheuristiques est l'introduction d'hyperparamètres supplémentaires, devant être également ajustés précisément, comme la taille de la population, l'opérateur de mutation, la stratégie de sélection, ou encore la valeur sélective d'un individu. De plus, en fonction du problème d'optimisation, les algorithmes métaheuristiques peuvent souffrir d'une convergence prématurée vers une solution sous-optimale. Similairement pour l'apprentissage profond, trouver le bon nombre de couches cachées, ainsi que le bon nombre de neurones pour chacune de ses couches, est si fastidieux, qu'il existe des techniques d'optimisation pour aider l'utilisateur à ajuster ces paramètres [YS20]. Pour répondre à ces problèmes, nous avons exploré plusieurs techniques récentes d'optimisation prometteuses pour essayer de les appliquer au domaine de l'injection de faute.

6.2.2 Contributions

De nouvelles techniques d'optimisation ont été proposées, comme les techniques d'optimisation bayésienne, ou les techniques d'optimisation du problème des bandits manchots pour explorer plus rapidement l'espace des paramètres. Bien que ces techniques ont déjà été utilisées pour l'optimisation d'hyperparamètres d'algorithmes d'apprentissage automatique, elles n'ont jamais été utilisée dans le cadre de l'injection de faute. Par conséquent, nous proposons d'utiliser, pour la première fois, un configurateur automatique basé sur l'optimisation bayésienne et un algorithme d'optimisation du problème des bandits manchots pour aider à identifier les meilleurs paramètres d'équipement. Également, nous proposons aussi de réduire la complexité de l'exploration de l'espace des paramètres en divisant l'optimisation en deux sous-étapes, mais contrairement à Carpi et al. [CPB+13] qui optimisent directement sur l'application ciblée, nous utiliserons un test de caractérisation de fautes. Ainsi nos contributions sons les suivantes :

- Nous appliquons pour la première fois deux techniques récentes d'optimisation d'hyperparamètres, pour trouver les meilleurs paramètres d'équipement de notre `GLITCH STATION`, pour trois microcontrôleurs 32-bit différents.
- Nous proposons de décomposer le problème d'optimisation en deux étapes, de manière à simplifier mais aussi à accélérer l'identification et l'exploitation de vulnérabilités en boîte noire. Tout d'abord, lors de l'*étape de caractérisation*, nous nous concentrons uniquement sur l'optimisation des paramètres d'équipement, puis, une fois les meilleures

configurations identifiées, lors de l'*étape d'exploitation*, nous déterminons les délais d'injection de faute pour attaquer l'application cible.

- En utilisant notre approche boîte noire avec les techniques d'optimisation proposées, nous contournons avec succès le mécanisme de protection de code d'un *bootloader* préchargé sur un microcontrôleur 32-bit, deux fois plus rapidement qu'avec des algorithmes génétiques.

6.3 Notre Approche

Cette section présente notre approche boîte noire pour optimiser l'exploitation de vulnérabilités sur microcontrôleurs. Notre stratégie vise à réduire au maximum le temps passé à chercher les meilleurs paramètres d'équipement, en réduisant la dimensionnalité de l'espace des paramètres. Accélérer l'exploration de l'espace des paramètres est particulièrement important du fait que les évaluations de sécurité sont souvent en temps contraint.

Comme nous l'avons précédemment évoqué, la stratégie commune est d'optimiser l'équipement de faute directement avec l'application ciblée. Dans ce cas, seule la section de l'application jugée critique par l'évaluateur, c'est-à-dire pouvant potentiellement conduire à des failles de sécurité, est testée. Cependant, identifier la section critique d'une application pouvant être très complexe est fastidieux, d'autant plus dans un contexte d'évaluation en boîte noire en temps contraint. Dans ces conditions, il est alors très difficile de trouver les réglages optimaux pour réussir une attaque par injection de fautes, pour les raisons déjà évoquées au chapitre précédent (chapitre 2).

6.3.1 Description de l'approche

Pour optimiser l'identification et l'exploitation de vulnérabilités à l'injection de fautes en boîte noire, nous proposons de réduire la dimensionnalité de l'espace des paramètres en divisant le problème d'optimisation en deux sous-étapes, de manière à simplifier et accélérer l'exploration de l'espace des paramètres. Tout d'abord, 1) l'*étape de caractérisation* optimise les paramètres de l'équipement indépendamment de l'application cible, en utilisant un test de caractérisation de fautes, et ensuite, avec les paramètres optimisés, 2), l'*étape d'exploitation*, identifie le délai d'injection de faute afin d'exploiter une vulnérabilité sur l'application cible. La figure 6.1 présente schématiquement notre approche. Si Carpi et al. [CPB+13] proposent une stratégie similaire en deux étapes, ils n'utilisent cependant pas de test de caractérisation de fautes ce qui, d'après le chapitre précédent (chapitre 5), peut engendrer des complications pour optimiser l'équipement d'injection de faute, en autres, une mauvaise propagation des fautes.

Test de caractérisation de fautes Comme pour l'approche en boîte grise, nous utilisons un ou plusieurs tests de caractérisation de manière à maximiser la propagation de fautes sur le microcontrôleur ciblé, afin de trouver plus rapidement les paramètres d'injection de faute. Tout d'abord, comme les tests sont constitués d'une série d'instructions répétée plusieurs fois,

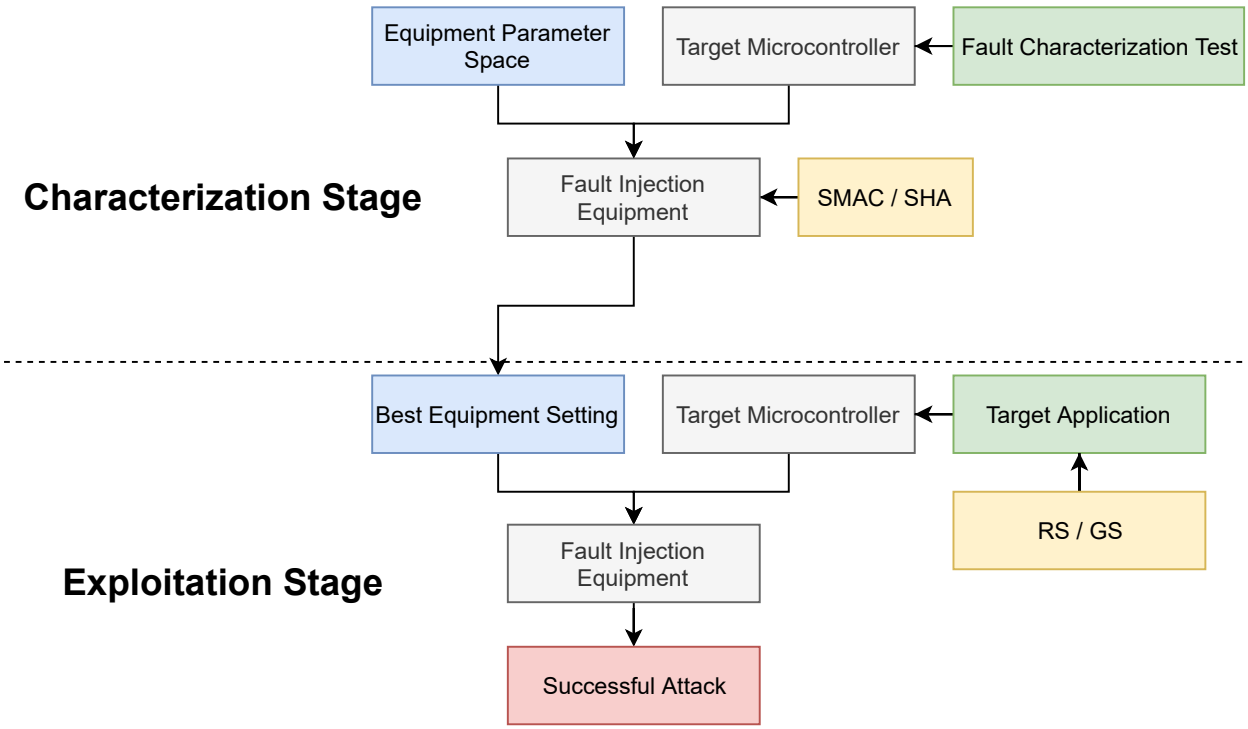


FIGURE 6.1 – Vue globale de notre approche pour optimiser l'identification et l'exploitation de vulnérabilités à l'injection de fautes en boîte noire.

le délai d'injection de faute peut être ignoré totalement pendant l'étape de caractérisation, permettant de réduire d'une dimension l'espace des paramètres. De plus, comme déjà détaillé au chapitre 2 et chapitre 5, le test de caractérisation de fautes est souvent plus court que l'application réelle ce qui permet de gagner du temps sur le long terme.

Techniques d'optimisation Nous utilisons différentes techniques d'optimisation pour chacune des étapes de notre approche. Pendant la phase de *caractérisation*, nous utilisons de nouvelles techniques d'optimisation d'hyperparamètres qui seront détaillées dans la section suivante, permettant d'explorer l'espace des paramètres bien plus rapidement que les techniques plus classiques comme GS ou RS. Ensuite, une fois les meilleures configurations identifiées, la recherche du délai d'injection pour exploiter une vulnérabilité peut se faire simplement avec une simple RS ou GS durant la phase d'exploitation. L'utilisation d'une technique d'optimisation plus complexe n'a pas de sens dans l'étape d'exploitation, car on ne cherche plus à optimiser l'équipement mais bien à exploiter une vulnérabilité.

6.4 Méthodes d'optimisation d'hyperparamètres

Cette partie est consacrée à la présentation des techniques d'optimisation, utilisées pour accélérer l'exploration de l'espace des paramètres d'un équipement d'injection faute donné, dans le but de trouver les meilleures configurations d'équipement. Tout d'abord, nous reve-

nons sur les particularités du problème d'optimisation des paramètres d'équipement. Puis nous détaillons les algorithmes proposés pour résoudre le problème des bandits manchots. Ensuite nous expliquons les récentes techniques d'optimisation bayésienne. Nous terminons en présentant de possibles alternatives aux techniques retenues.

6.4.1 Problème d'optimisation des paramètres d'équipement

Certains problèmes d'optimisation sont plus simples que d'autres. Lorsque la fonction-objectif à minimiser est convexe, différentiable, et qu'elle admet un minimum global, n'importe quelle méthode basée sur une descente en gradient trouvera la solution. Néanmoins, notre problème d'optimisation n'appartient pas à cette catégorie, du fait de plusieurs particularités propres à ce dernier.

Optimisation stochastique L'une des premières difficultés dans la résolution du problème d'optimisation des paramètres d'équipement tient du caractère non-déterministe des fautes injectées. Il ne suffit pas de tester une seule fois une configuration d'équipement pour évaluer sa performance. Au contraire, plusieurs essais sont nécessaires pour estimer empiriquement la *probabilité de faute* pour chaque configuration.

Optimisation sans dérivées Les dérivées partielles de la fonction-objectif par rapport aux paramètres d'équipement ne sont pas accessibles, par conséquent, les approches classiques par descente de gradient ne sont pas applicables.

Optimisation discrète Le plus souvent, les paramètres des équipements d'injection de faute prennent des valeurs discrètes. Toutes les techniques d'optimisation ne sont pas adaptées à traiter des variables discrètes.

Optimisation avec contraintes L'espace des paramètres peut être contraint pour ne pas endommager la cible ou pour viser uniquement une zone délimitée (e.g. la mémoire Flash). Certaines techniques d'optimisation ne prennent pas en compte les problèmes avec contraintes.

Optimums locaux multiples La fonction-objectif n'est pas convexe, et plusieurs paramètres d'équipement d'injection de faute peuvent être localement optimaux. Cela peut conduire à une convergence prématurée vers une solution sous-optimale.

Pas de parallélisation Il est difficilement possible de paralléliser le problème, car cela nécessiterait plusieurs bancs de test pour conduire les expérimentations en parallèle.

Coût d'évaluation Si tester une nouvelle configuration d'équipement est rapide, il y a toujours un risque de détériorer le microcontrôleur cible pendant les expérimentations, sachant que la préparation d'une cible peut aller de quelques heures à quelques jours en fonction de la technique d'injection de faute utilisée.

D'autre part, l'espace des paramètres d'équipement est généralement trop grand pour une recherche exhaustive. La recherche par quadrillage et la recherche aléatoire, bien que souvent utilisées, ne sont pas optimales pour explorer efficacement l'espace (chapitre 2). Ainsi, nous nous sommes int éressés à d'autres techniques d'optimisation adaptées aux particularités de notre problème, et en particulier, *Successive Halving Algorithm* (SHA) et *Sequential Model-Based Algorithm Configuration* (SMAC).

6.4.2 SHA

6.4.2.1 Introduction au problème des bandits manchots

L'algorithme de réduction de moitié successive, en anglais *Successive Halving Algorithm (SHA)* a été proposé pour la première fois par Karnin et al. [KKS13] en 2013 pour résoudre le problème des bandits manchots.

En théorie des probabilités, le problème des bandits manchots est un problème dans lequel un ensemble fixe de ressources limitées, aussi appelé budget, doit être alloué entre plusieurs choix (aussi désigné sous le terme de *bras*). Toute la difficulté du problème tient du fait que ces choix semblent équivalents de prime abord, car la récompense moyenne, propre à chacun des choix, n'est pas connue a priori, et ce n'est qu'en allouant des ressources à un choix qu'une meilleure connaissance sur cette dernière pourra être acquise [BF85] ; l'objectif étant de maximiser le gain cumulé pour le budget alloué.

On dit encore que l'objectif est de minimiser le *regret*, c'est-à-dire la différence entre la récompense cumulée en choisissant systématiquement le meilleur bras n fois et la récompense cumulée des n bras sélectionnés. Le regret R après n essais peut se définir formellement de cette manière [Sli19], où $\mu^\star$ est la récompense associée au meilleur bras possible et $\mu(b_k)$ est la récompense obtenue en choisissant le bras b à l'essai k :

$$R(n) = n\mu^\star - \sum_{k=1}^{n} \mu(b_k) \tag{6.1}$$

L'illustration classique de ce problème, à l'origine du nom par ailleurs, est d'imaginer un parieur dans un casino, qui devant des machine à sous (aussi appelé *bandit manchot*), doit décider lesquelles jouer, dans le but de maximiser ses gains avec son budget limité.

C'est un problème classique d'apprentissage par renforcement, et par conséquent, un parfait exemple du dilemme entre exploration et exploitation. On est constamment confronté à choisir d'allouer des ressources au bras avec la meilleure récompense moyenne (exploitation), ou au contraire, allouer des ressources pour acquérir de meilleures connaissances sur les récompenses moyennes des autres choix (exploration).

Si on retrouve facilement des analogies avec notre problème d'exploration de l'espace des paramètres d'un équipement d'injection de faute donné, il ne faut pas oublier que le problème des bandits manchots dans sa forme la plus classique considère un nombre discret de bras, inférieur au budget alloué, ce qui, dans notre situation, ne convient pas, du fait que la taille de l'espace des paramètres est généralement bien supérieure au nombre d'injections réalisables en temps contraint. Cependant, il existe plusieurs variantes du problème des bandits manchots, y compris lorsque le nombre de bras à considérer est largement supérieur

au budget maximal qu'il est possible d'allouer [BCZ+97]. En relâchant la contrainte sur le nombre de bras considérés, cette variante du problème est alors transposable à notre problème initial d'exploration de l'espace des paramètres.

Dans le paragraphe suivant, nous décrivons une variante du SHA, applicable à notre problématique.

6.4.2.2 Description de l'algorithme

L'algorithme SHA, proposé par Karnin et al. [KKS13], est déjà utilisé dans le cadre de l'optimisation des hyperparamètres de modèles d'apprentissage automatique [YS20]. L'Algorithme 3 détaille l'algorithme de réduction de moitié successive.

Le but de cet algorithme est d'identifier le meilleur bras (ici la meilleure configuration) dans les limites d'un budget fixe T (ici le nombre total d'injections de fautes). Le budget total fixé est alloué uniformément sur $log_2(n)$ tours d'élimination, où n est le nombre de configurations initiales $\vec{\Theta}_0$ échantillonnées de l'espace des paramètres Θ. L'algorithme évalue ensuite chaque configuration de manière uniforme. À la fin de chaque tour, les pires configurations sont éliminées. Ensuite, au tour suivant, les configurations restantes sont évaluées deux fois plus qu'au tour précédent et ainsi de suite jusqu'à ce qu'il ne reste plus que la meilleure configuration θ_{inc}.

Algorithme 3 : SHA

Input : Budget total fixé T, l'espace de paramètre d'injection de faute Θ, n configurations intiales $\vec{\Theta}_0 \subset \Theta$

Output : La meilleure configuration $\theta_{inc} \in \vec{\Theta}_{\lceil \log_2(n) \rceil}$

for $r = 0$ **to** $\lceil \log_2(n) \rceil - 1$ **do**

- $t_r \leftarrow \left\lfloor \frac{T}{|\vec{\Theta}_r| \lceil \log_2(n) \rceil} \right\rfloor$;
- **foreach** $\theta_i \in \vec{\Theta}_r$ **do**
  - Test t_r times each configuration θ_i;
  - Compute the empirical mean $\mu_{r,i}$ of θ_i;
- $k_r \leftarrow \lceil |\vec{\Theta}_r|/2 \rceil$;
- `/* Keep the` k_r^{th} `best` θ_i `with the largest` $\mu_{r,i}$ `*/`
- $\vec{\Theta}_{r+1} \leftarrow$ `BestKthConfigurations(`$\vec{\Theta}_r, k_r$`)`;

return $\theta_{inc} \in \vec{\Theta}_{\lceil \log_2(n) \rceil}$;

Dans la version proposée par Karnin et al., pour un budget total fixé T, toute la difficulté réside dans le choix du nombre n de configurations initiales. On peut prendre n grand, entrainant l'évaluation d'un grand nombre de configurations différentes, mais réduisant le nombre d'essais consacrés pour chacune d'entre elles, ou bien prendre n petit, permettant d'augmenter le nombre d'essais pour évaluer chaque configuration, au détriment du nombre de configurations évaluées.

La solution proposée par Aziz [Azi19] consiste à choisir n, puis à prendre un budget $T = nlog_2(n)$, ce qui entraine une sélection agressive des configurations après seulement un essai,

dès le premier tour ($t_0 = 1$). Bien que seule une conjecture sur la borne supérieure de regret a été proposée, l'algorithme de réduction de moitié successive avec la paramétrisation $T = nlog_2(n)$ obtient empiriquement des résultats similaires, si ce n'est meilleurs, que d'autres solutions plus complexes, telles que HyperBand [LJD+17]. Nous reviendrons sur HyperBand à la sous-section 6.4.4.

Néanmoins, le fait de sélectionner agressivement les configurations après seulement un seul essai peut être une limitation contraignante, particulièrement dans le cas d'une évaluation de microcontrôleur sécurisé. En effet, dans ce cas précis, il peut être difficile d'obtenir des fautes. Il est d'ailleurs fort probable de ne pas réussir à fauter le microcontrôleur lors des premières itérations avec les configurations testées. La sélection se fait alors aléatoirement car aucune configuration n'est jugée meilleure. Les résultats finaux seront alors potentiellement inutilisables.

Ceci étant dit, nous verrons à la section 6.5 que SHA permet d'identifier des paramètres d'équipement plus performants qu'avec les autres techniques d'optimisation.

6.4.3 SMAC

Le configurateur automatique d'algorithmes par optimisation séquentielle basée sur un modèle, proposé par Hutter et al. en 2011 [HHLB11], en anglais *Sequential Model-Based Algorithm Configuration* (SMAC), est un configurateur automatique d'algorithmes, qui comme le nom l'indique, s'appuie sur l'optimisation séquentielle basée sur un modèle, en anglais *Sequential Model-Based Optimization* (SMBO), qui est une généralisation de l'optimisation bayésienne.

SMAC a déjà été appliquée pour optimiser les hyperparamètres de solveurs complexes de programmation par contraintes (e.g. IBM CPLEX et ses 76 paramètres [HHLB11]) ou bien des modèles d'apprentissage automatique [YS20]. Cependant, contrairement aux approches classiques basées sur l'optimisation bayésienne, SMAC supporte tout type de paramètres, y compris continus, discrets et qualitatifs, mais supporte également les processus non-déterministes ce qui est extrêmement important dans le cadre de l'injection de fautes, comme vu précédemment. Le paragraphe suivant présente succinctement l'optimisation séquentielle basée sur un modèle.

6.4.3.1 Optimisation séquentielle basée sur un modèle

Contrairement aux approches précédentes (SHA, GA, GS, RS), la principale caractéristique de SMBO est de tenir compte des résultats précédents pour adapter itérativement un modèle probabiliste, dans le but de sélectionner plus intelligemment les prochaines configurations d'équipement pouvant potentiellement maximiser le nombre de fautes sur le microcontrôleur. SMBO, comme détaillé dans Algorithme 4, est un algorithme itératif, articulé autour de deux éléments clés que l'on retrouve aussi en optimisation bayésienne classique, à savoir un *modèle probabiliste* et une *fonction d'acquisition*, aussi appelés modèle de substitution et fonction de sélection, respectivement. Le mécanisme d'*intensification* n'existe quant à lui pas dans la version classique de l'optimisation bayésienne.

Modèle probabiliste Comme la fonction-objectif n'est pas connue, un modèle probabiliste, ou modèle de substitution, généré à partir des données observables, va permettre d'approximer cette dernière. Le modèle probabiliste $\mathcal{M}$ est adapté (`FitModel`) aux résultats précédents $\mathbf{R} = \{(\theta_1, o_1), ..., (\theta_n, o_n)\}$ où θ_i est une configuration de l'espace des paramètres considérés Θ, et o_i est la probabilité de faute observée avec la configuration θ_i. Le modèle $\mathcal{M}$ vise à prédire la probabilité de faute o_{i+1} pour une nouvelle configuration θ_{i+1} afin de savoir si θ_{i+1} vaut la peine d'être évaluée expérimentalement.

Fonction d'acquisition Les nouvelles configurations $\vec{\Theta}_{new}$ à évaluer pour l'itération courante sont sélectionnées de l'espace de paramètre d'injection de faute Θ par la fonction d'acquisition (`SelectConfigurations`), qui conserve un équilibre entre *exploitation* (échantillonner directement dans l'espace où le modèle prédit la probabilité de faute la plus élevée) et *exploration* (échantillonner là où le modèle n'a pas de distribution de probabilité *a priori*).

Intensification En plus du modèle probabiliste et de la fonction d'acquisition, SMBO ajoute un processus important *d'intensification* (`Intensify`) qui détermine 1) le budget alloué pour chaque configuration θ_i (i.e. le nombre d'essai pour chaque configuration), et 2) la meilleure configuration connue pour le moment θ_{inc}. C'est ce mécanisme qui permet notamment de tenir compte du caractère non-déterministe des fautes en laissant la possibilité de réévaluer plusieurs fois la même configuration. Aussi, le budget alloué à chaque configuration est adapté en fonction des résultats, et une nouvelle configuration est rejetée dès lors qu'elle sous-performe la meilleure configuration connue [HHLB11].

Algorithme 4 : Optimisation séquentielle basée sur un modèle

Input : Budget total fixé T, l'espace de paramètre d'injection de faute Θ, les configurations initiales $\vec{\Theta}_{init} \subset \Theta$

Output : La meilleure configuration θ_{inc}

$\mathbf{R}, \theta_{inc} \leftarrow$ `Initialize`$(\vec{\Theta}_{init})$;
repeat
  /* Fit the model $\mathcal{M}$ based on results $\mathbf{R}$ */
  $\mathcal{M} \leftarrow$ `FitModel`$(\mathbf{R})$;
  /* Select promising configurations $\vec{\Theta}_{new}$ */
  $\vec{\Theta}_{new} \leftarrow$ `SelectConfigurations`$(\mathcal{M}, \Theta)$;
  /* Find the best configuration θ_{inc} */
  $\mathbf{R}, \theta_{inc} \leftarrow$ `Intensify`$(\theta_{inc}, \vec{\Theta}_{new})$;
until *total budget T is exhausted*;
return θ_{inc};

Si SMBO est une généralisation de l'optimisation bayésienne, SMAC peut être vu comme une instanciation de SMBO, avec pour modèle probabiliste une forêt d'arbres décisionnels, et pour fonction d'acquisition l'Amélioration Attendue ; ce que nous détaillerons dans le prochain paragraphe.

6.4.3.2 SMAC

SMAC est un configurateur automatique d'algorithmes basé sur l'optimisation bayésienne, reposant sur un modèle probabiliste original, une forêt d'arbres décisionnels. La fonction d'acquisition, l'Amélioration Attendue, est quant à elle plus classique et s'adapte aussi bien aux modèles probabilistes basés sur une forêt d'arbres décisionnels qu'à un processus gaussien.

Forêt d'arbres décisionnels SMAC s'appuie sur une forêt d'arbres décisionnels, aussi appelée *Random Forest* (RF) pour son modèle de substitution. Le modèle probabiliste choisit par SMAC est radicalement différent des modèles plus classiquement utilisés comme, par exemple, le modèle de processus Gaussien. C'est ce choix atypique qui permet à SMAC de supporter les paramètres continus, discrets et qualitatifs. RF [Bre01] est une méthode ensembliste combinant le résultats de plusieurs arbres décisionnels pour résoudre des problèmes de classification ou bien de régression. Concernant cette dernière classe de problème, les arbres décisionnels prennent au niveau de leurs feuilles des valeurs continues plutôt que des labels, et sont communément appelés des arbres de régression. SMAC estime la performance (ici la probabilité de faute) moyenne μ_θ et la variance σ_θ^2 pour une nouvelle configuration θ, en calculant la moyenne et la variance empirique de la probabilité de faute estimée avec chaque arbre de régression de la forêt. Chaque arbre de régression de la forêt est généré à partir d'un sous ensemble des observations précédentes $\mathbf{R}$. Cette étape, aussi appelée *bagging*, consiste à tirer aléatoirement n observations avec remise directement dans l'ensemble des résultats précédent $\mathbf{R}$. Ensuite, pour chaque nœud, m critères (e.g. les paramètres d'injection de fautes), sont sélectionnés aléatoirement à partir des critères initiaux, et celui qui (parmi ceux déjà sélectionnés) minimise la somme des carrés des résidus est choisi pour séparer le nœud en deux branches. Le processus général est résumé à la figure 6.2.

Amélioration Attendue La fonction d'acquisition de SMAC est l'Amélioration Attendue, aussi appelée *Expected Improvement* (EI), une fonction que l'on retrouve couramment en optimisation bayésienne. Cette fonction est utilisée pour quantifier à quel point une nouvelle configuration θ pourrait améliorer la performance (la probabilité de faute) par rapport à la meilleure configuration connue θ_{inc} pour le moment. Comme la fonction-objectif est inconnue, EI est calculé en utilisant la distribution de probabilité *a posteriori* de θ connaissant la performance moyenne μ_θ et la variance σ_θ^2 prédites par le modèle probabiliste (RF pour SMAC) et la performance moyenne empirique de la meilleure configuration connue θ_{inc}. Pour plus d'information sur l'amélioration attendue et son implémentation dans SMAC, se référer aux travaux de Hutter et al. [HHLB11, HHLBM09]. Finalement, les configurations maximisant l'Amélioration Attendue sont sélectionnées pour être évaluées expérimentalement. La figure 6.3 illustre un exemple où l'amélioration attendue permet de sélectionner les prochaines configurations à évaluer à l'aide d'un modèle de processus gaussien.

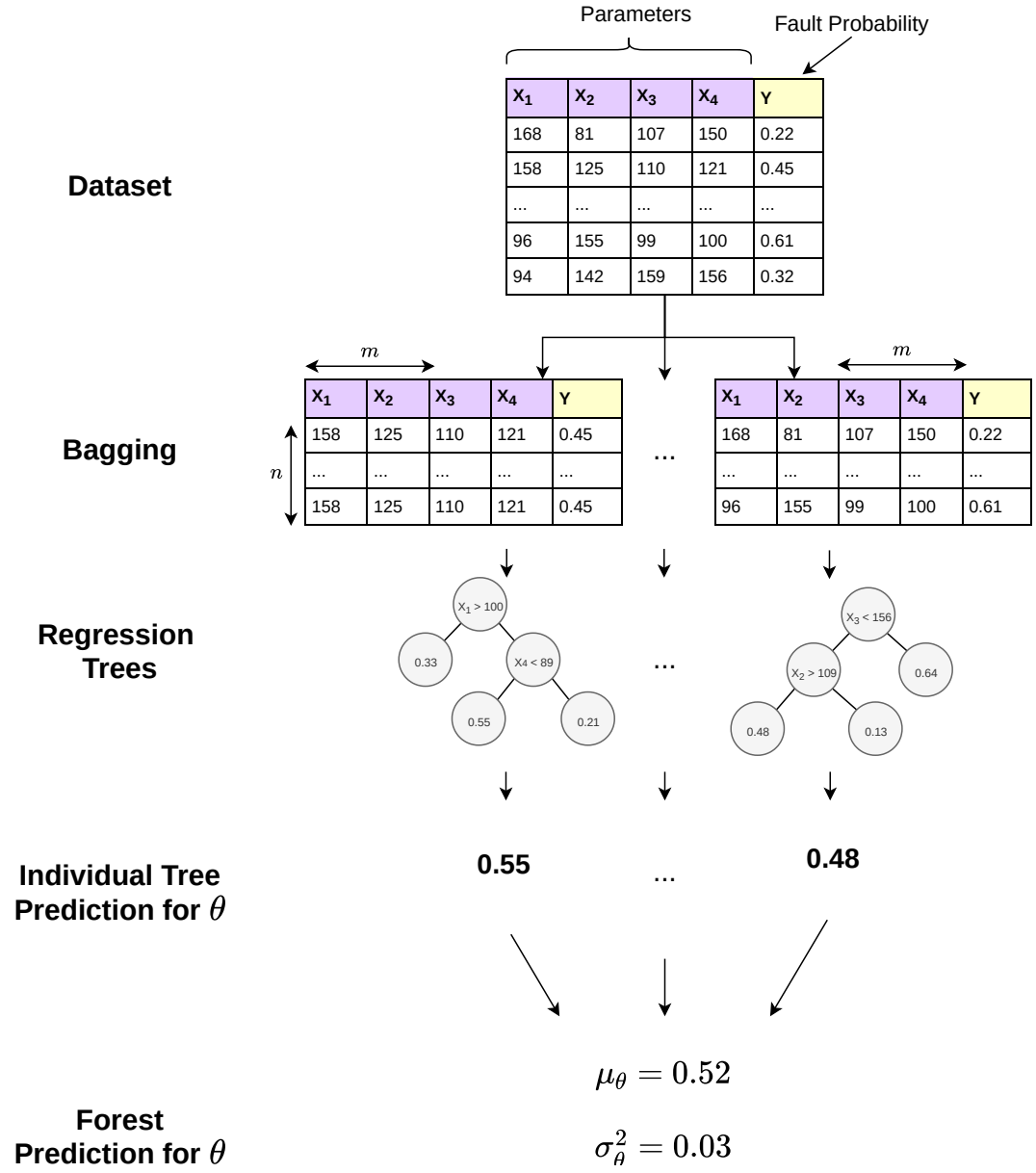


FIGURE 6.2 – Création d'une forêt d'arbres de régression via *bagging* et génération du modèle probabiliste de la fonction-objectif. À partir des résultats précédents (*Dataset*), on sélectionne n observations avec remise. Ensuite, B arbres de régression sont formés. À chaque nœud, on sélectionne m paramètres. Le paramètre qui minimise la somme des carrés des résidus est choisi pour séparer le nœud en deux branches. La performance d'une nouvelle configuration θ est estimée en prenant la moyenne des prédictions des arbres de la forêt.

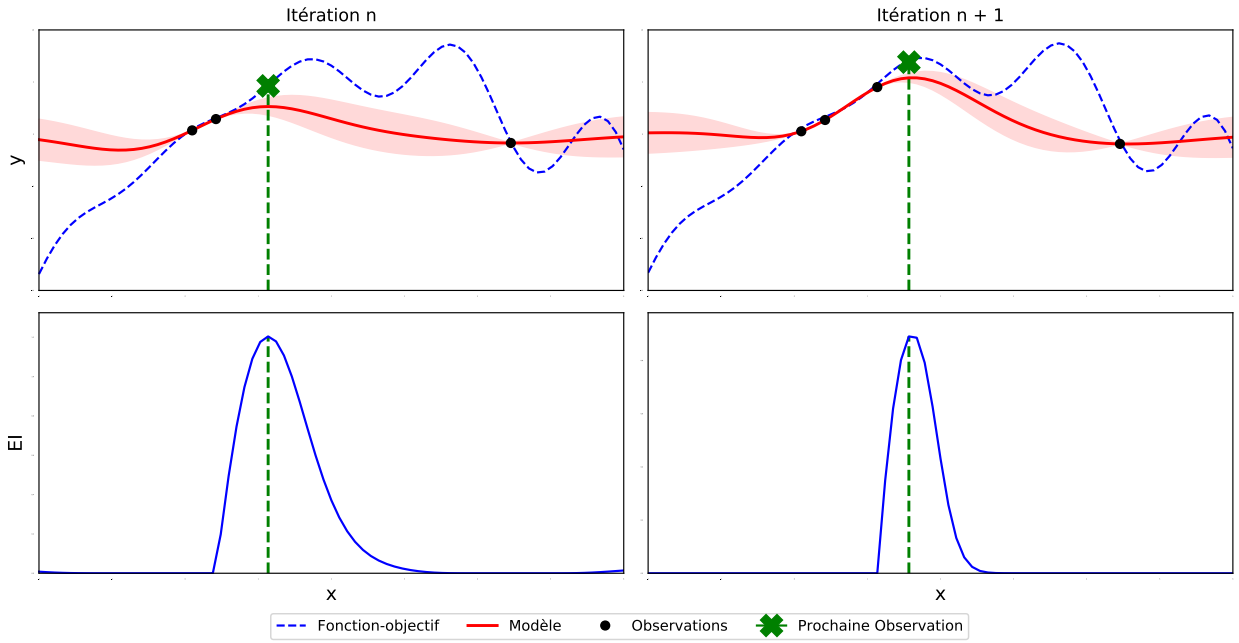


FIGURE 6.3 – Sélection du prochain point à tester pour l'itération suivante via l'Amélioration Attendue (EI), en fonction de la distribution *a posteriori* (modèle) construite à partir des observations.

6.4.3.3 SMAC dans le contexte de l'injection de fautes

Si SMAC est un outil de configuration automatique très puissant ayant déjà fait ses preuves pour optimiser des solveurs complexes de programmation par contraintes [HHLB11], une des limitations principales de SMAC est que les conditions initiales affectent grandement les performances en termes de vitesse de convergence. Par conditions initiales, nous faisons référence au sous-ensemble de configurations initiales $\vec{\Theta}_{init}$, qui sont, par défaut, tirées aléatoirement à partir de l'espace des paramètres d'injection de fautes Θ.

Lorsque le microcontrôleur ciblé est difficile à fauter, et qu'aucune faute n'est obtenue, par exemple dans le cas de microcontrôleurs sécurisés, SMAC va peiner à explorer rapidement et correctement l'espace des paramètres d'injection de fautes. Cela s'explique du fait que, n'ayant pas de référentiel sur l'objectif à accomplir (toutes les configurations ont une probabilité de faute nulle), SMAC n'arrivera pas à adapter le modèle, et encore moins à sélectionner des configurations pertinentes. Pour combler cette limitation, nous avons rajouté une procédure préliminaire en deux étapes, pour être sûr d'avoir réussi à fauter au moins une fois le composant :

- *Exploration pure* : des configurations $\theta \in \Theta$ sont tirées aléatoirement et évaluée expérimentalement jusqu'à ce que 1) au moins k_{min} configurations induisant des fautes soient trouvées, **et** 2) n_{min} fautes ont été injectées. Par défaut, $k_{min} = 1$ et $n_{min} = 1000$.

- *Mutation* : le sous-ensemble de configurations initiales $\vec{\Theta}_{init}$, contenant au moins k_{min} configurations identifiées pendant l'étape d'exploration pure, est complété avec des

	$s=4$		$s=3$		$s=2$		$s=1$		$s=0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	27	3	9	9	6	27	5	81
1	27	3	9	9	3	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								

TABLE 6.1 – Détail des brackets alloués par HyperBand [LJD+17].

configurations additionnelles, générées en utilisant un operateur de mutation gaussien [BS02] à partir des configurations existantes, et ce jusqu'à atteindre $|\vec{\Theta}_{init}| = k_{init}$ configurations. Par défaut, $k_{init} = 100$.

En fonction du microcontrôleur cible, k_{min}, n_{min} et k_{init} doivent être ajustés. Par exemple, dans le cas d'un microcontrôleur peu sensible aux fautes, étendre la phase d'exploration pure (i.e. $k_{min} > 1$ et $n_{min} > 1000$) peut aider significativement SMAC en lui donnant grossièrement les zones de l'espace à explorer en priorité.

6.4.4 Technique d'optimisation alternatives

D'autres techniques que SMAC ou SHA pourraient potentiellement être utilisées pour optimiser les paramètres d'équipements, et n'ont pas été évaluées pour le moment en pratique. Nous présentons brièvement deux d'entre elles, *HyperBand* et *Tree-structured Parzen Estimator*.

6.4.4.1 HyperBand

HyperBand est un algorithme qui se base sur plusieurs itérations successives de l'algorithme SHA présenté précédemment. HyperBand propose de résoudre le problème principal du SHA, à savoir le dimensionnement du nombre n de configurations initiales et d'essai r, en les allouant dynamiquement à chaque itération. Pour cela, le budget total alloué est divisé équitablement sur plusieurs itérations de SHA, appelé *bracket*. À la fin de chaque itération de SHA, les meilleures configurations sont passées à l'itération suivante.

Les données du tableau 6.1 proviennent d'un exemple du papier original présentant HyperBand [LJD+17]. On retrouve pour chaque bracket $s = i_{max}$ d'HyperBand le nombre de configurations testées n_i et d'essais r_i en fonction du tour i.

Les brackets sont de moins en moins agressifs après chaque itération. Autrement dit, le premier bracket $s = 4$ favorise l'exploration en sélectionnant après seulement un essai, tandis que le dernier $s = 0$ favorise l'exploitation en testant seulement 5 configurations. Si l'idée proposée par HyperBand est intéressante, on peut se demander si ce dernier est significativement meilleur qu'un SHA classique, présenté précédemment.

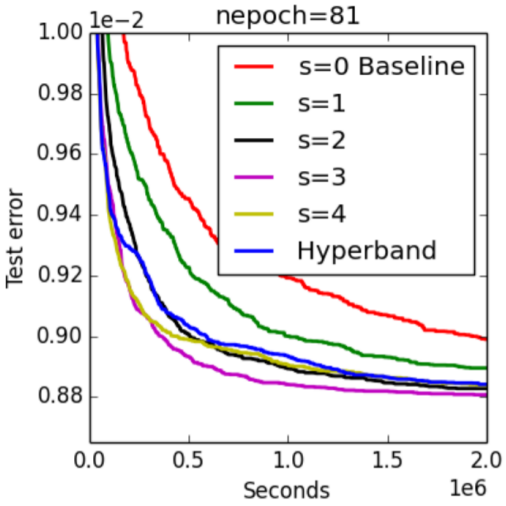


FIGURE 6.4 – Performance des brackets individuels et HyperBand [LJD+17].

Le papier original compare les performances d'HyperBand aux performances individuelles de chaque bracket à la figure 6.4. On observe que le *bracket* le plus agressif dans la sélection des configurations ($s = 4$) obtient des performances similaires à HyperBand. Ces résultats nous ont poussés à choisir la solution de Aziz [Azi19] (T=$nlog_2(n)$) plutôt qu'HyperBand, ce dernier étant plus complexe et n'apportant pas de gain en performance.

Néanmoins, il pourrait être intéressant d'évaluer HyperBand en pratique avec notre problème d'optimisation de paramètres d'équipement et comparer les performances avec un SHA classique.

6.4.4.2 Tree-structured Parzen Estimator

L'optimisation bayésienne basée sur la méthode d'estimation par noyau Parzen-Rosenblatt (BO-TPE), proposée par Bergstra et al [BBBK11], partage de nombreuses similitudes avec SMAC. Cependant,la principale différence entre SMAC et BO-TPE est l'approche choisie pour modéliser la fonction-objectif.

Si SMAC modélise $P(o|\theta)$, c'est-à-dire l'estimation de la probabilité de faute o qui sera observée sachant la configuration d'équipement θ utilisée, BO-TPE, quant à lui, modélise $P(\theta|o)$ et $P(o)$. Plus précisément, BO-TPE modélise la probabilité des configurations suivant qu'elles améliorent ou non la probabilité de faute, selon un quantile $o*$ fixé. En effet, BO-TPE partitionne les probabilités de faute observées en deux groupes $\mathcal{L}$ et $\mathcal{G}$, selon qu'elles soient inférieures ou supérieures au quantile $o*$ respectivement (figure 6.5). Les fonctions de densité $l(\theta)$ et $g(\theta)$ associées aux groupes $\mathcal{L}$ et $\mathcal{G}$ sont ensuite estimées via la méthode Parzen-Rosenblatt.

$$P(\theta|o) = \begin{cases} l(\theta) & \text{si } o < o* \\ g(\theta) & \text{si } o \geq o* \end{cases}$$

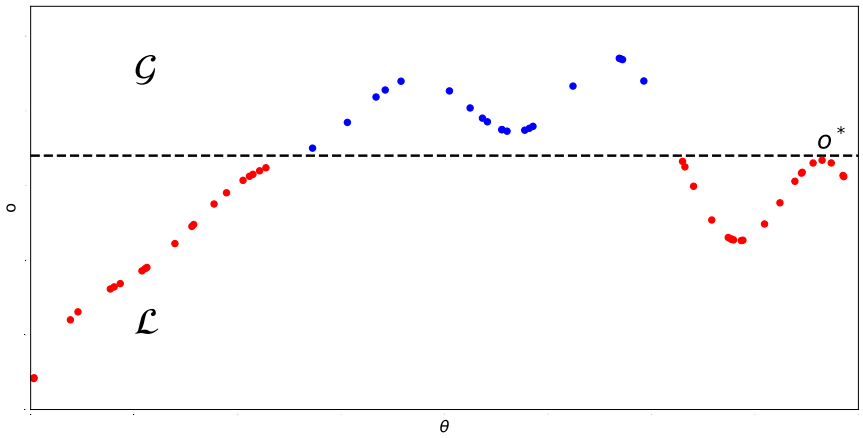


FIGURE 6.5 – BO-TPE partitionne les probabilités de faute observées en deux groupes $\mathcal{L}$ et $\mathcal{G}$ selon le quantile o^*.

Enfin, l'Amélioration Attendue est adaptée aux spécificités de BO-TPE [BBBK11] : les prochaines configurations sélectionnées sont celles maximisant $EI = \frac{g(\theta)}{l(\theta)}$.

Comme pour SMAC, BO-TPE supporte tous types de paramètres, y compris continus, discrets et qualitatifs, ainsi que les processus non-déterministes. Par conséquent, si HyperBand nous a laissé sur notre faim, BO-TPE est sans doute une piste très intéressante qui mériterait d'être creusée. Pour information, une implémentation de BO-TPE est proposée dans la librairie HyperOpt [BYC+13].

Après cette courte parenthèse sur les techniques d'optimisation alternatives à celles proposées, nous nous intéresserons, dans la partie suivante, aux performances en pratique des techniques d'optimisation SMAC et SHA.

6.5 Évaluation des techniques d'optimisation en pratique

Dans cette section, nous allons optimiser notre `GLITCH STATION` pour trois microcontrôleurs 32-bit différents, en utilisant SMAC, SHA, GA et RS. Nous allons montrer que les techniques d'optimisation présentées précédemment, à savoir SHA et SMAC, permettent d'identifier plus rapidement les meilleures configurations que les techniques habituellement utilisées. Après avoir présenté les microcontrôleurs cibles et leurs principales caractéristiques, nous détaillerons le protocole expérimental. Enfin, nous comparons la performance (probabilité de faute et vitesse de convergence) de SMAC et SHA avec les techniques plus communément utilisées, GA et RS.

6.5.1 Microcontrôleurs cibles

Nous avons sélectionné pour cette expérience trois microcontrôleurs 32-bit différents, basés sur différents cœurs ARM Cortex-M. L'architecture et les caractéristiques de ces com-

posants varient selon le modèle considéré, et par conséquent ces microcontrôleurs ne réagiront pas de la même manière aux injections de faute par perturbation de la tension d'alimentation. Les microcontrôleurs sélectionnés sont :

- **µC-M0** est un microcontrôleur faible consommation basé sur le cœur ARM Cortex-M0+ cadencé à 24Mhz, reposant sur une architecture Von Neumann ARMv6-M avec un *pipeline* de deux étages (chargement et exécution).
- **µC-M3** est un microcontrôleur populaire reposant sur le cœur ARM Cortex-M3 également cadencé à 24Mhz, implémentant une architecture Harvard ARMv7-M avec un *pipeline* de trois étages (chargement, décodage, exécution).
- **µC-M4** est un microcontrôleur ultra basse consommation basé sur un cœur ARM Cortex-M4 cadencé à 72Mhz, reposant sur une architecture Harvard ARMv7E-M avec un *pipeline* de trois étages (chargement, décodage, exécution) et prédiction de branchement.

6.5.2 Protocole Expérimental

Information Générale À l'étape de caractérisation, nous avons utilisé notre ensemble minimal de test, et plus particulièrement le test de caractérisation de fautes IC détaillé à la table (tableau 5.12). Ce test a été conçu de manière à maximiser la propagation des bit-set ou des bit-reset sur l'instruction chargée, mais aussi le saut d'instruction, comme détaillé au chapitre 5. Les autres tests (RC et MC) propagent significativement moins de fautes pour les microcontrôleurs évalués et n'ont pas été retenus. Pour chaque technique d'optimisation (SMAC, SHA, GA et RS), nous avons injecté 50.000 fautes ($\approx$ 6 heures). Pour SMAC, nous avons utilisé la libraire Python SMACv3 [LEF+17], et plus précisément la classe `SMAC4HPO`. Pour SHA, GA et RS, nous n'avons pas utilisé de libraire externe, ces algorithmes étant moins complexes. Pour SHA, comme décrit précédemment, nous avons utilisé la paramétrisation $T = nlog_2(n)$, avec $n = 4096$. Pour GA, chaque individu de la population représente une configuration valide de l'équipement d'injection de fautes considéré. Nous avons entrainé une population de 50 individus sur 200 générations, où chaque individu a été testé 5 fois. Après les essais, on compte le nombre de résultats fautés obtenus avec la configuration testée, puis on estime la valeur sélective de la configuration avec $\frac{\#\{\text{résultats fautés}\}}{\#\{\text{essais}\}}$. De plus, nous avons utilisé un opérateur de mutation gaussien [BS02], une sélection par tirage à la roulette avec acceptation stochastique (*roulette-wheel selection via stochastic acceptance*) [LL12], et la valeur sélective (*fitness*) d'un individu (configuration) est déterminée par sa probabilité de faute. Pour RS, nous évaluons 10.000 configurations, où chaque configuration est testée cinq fois. Les paramétrisations des techniques d'optimisation évaluées sont résumées au tableau 6.2.

Protocole Les résultats de l'optimisation de l'équipement d'injection de fautes avec SMAC, SHA, GA et RS sont hétérogènes. Tandis que SMAC et SHA, de par leur conception, retournent seulement une seule configuration (la meilleure trouvée), RS et GA retournent

Technique	Paramétrisation
RS	10.000 configurations testées 5 essais par configuration
GA	200 générations 50 configurations par génération 5 essais par configuration Opérateur de mutation gaussien Tirage à la roulette avec acceptation stochastique Valeur sélective d'une configuration $\equiv \frac{\#\{\text{résultats fautés}\}}{\#\{\text{essais}\}}$
SHA	$T = nlog_2(n)$ $n = 4096$
SMAC	$T = 50000$ $k_{min} = 1$ $n_{min} = 1000$ $k_{init} = 100$

TABLE 6.2 – Paramétrisation des techniques d'optimisation évaluées.

plusieurs configurations. En effet, SMAC et SHA augmentent progressivement le nombre d'essais pour chaque configuration testée, de manière à approximer plus précisément la probabilité de faute afin de sélectionner la meilleure, alors que RS et GA évaluent toujours chaque configuration le même nombre de fois, et ainsi, plusieurs configurations peuvent avoir la même probabilité de faute. Par conséquence, pour comparer équitablement l'évolution de la probabilité de faute au cours du temps des configurations trouvées avec SMAC, SHA, GA et RS, plusieurs considérations doivent être prises en compte :

- *SMAC* : de part sa conception, SMAC met à jour automatiquement la meilleure configuration connue pour le moment pendant l'exécution, et donc aucun post-traitement n'est requis.
- *RS* : contrairement à SMAC, un post-traitement sur les résultats est requis. Toutes les 5000 injections de fautes, nous injectons 1000 de plus pour évaluer la probabilité de faute de la meilleure configuration trouvée pour le moment.
- *GA* : Le même post-traitement sur les résultats que RS est requis.
- *SHA* Nous évaluons la probabilité de faute moyenne des configurations restantes après chaque réduction successive de moitié.

Pour chaque microcontrôleur considéré, nous optimisons notre `GLITCH STATION` en utilisant SMAC, SHA, GA et RS et nous comparons l'évolution de probabilité de faute des meilleures configurations trouvées au cours du temps. La meilleure technique d'optimisation

		SMAC	SHA	GA	RS
µC-M0	Probabilité de faute Max	0.52	**0.53**	0.49	0.49
	Vitesse de convergence	**Rapide**	Lente	**Rapide**	**Rapide**
µC-M3	Probabilité de faute Max	0.77	**0.81**	0.52	0.24
	Vitesse de convergence	**Rapide**	Lente	Lente	Lente
µC-M4	Probabilité de faute Max	**0.95**	0.79	0.81	0.71
	Vitesse de convergence	**Rapide**	Lente	**Rapide**	**Rapide**

TABLE 6.3 – Comparaison de la performance entre les techniques d'optimisation.

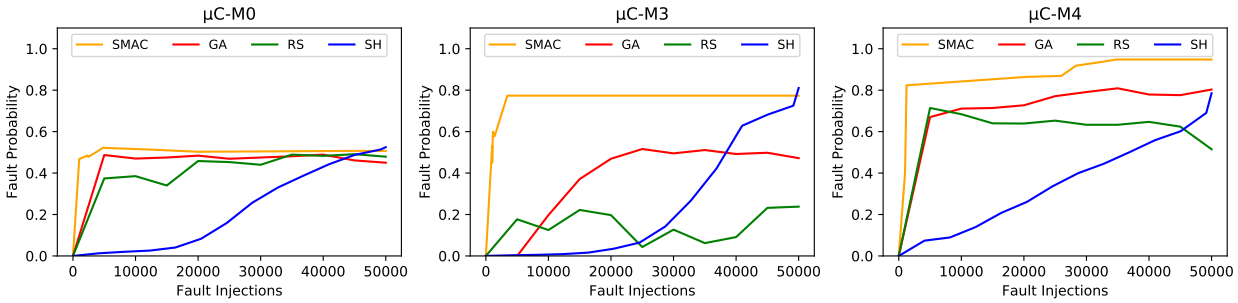


FIGURE 6.6 – Évolution de la probabilité de faute sur 50.000 injections de fautes, en utilisant SMAC, GA, SHA and RS, avec notre `GLITCH STATION`

est celle qui trouve les configurations avec la plus haute probabilité de faute, en un minimum d'injection de fautes.

6.5.3 Résultats

Les résultats des expériences sont résumés à la figure 6.6 et au tableau 6.3. Dans la figure 6.6, nous comparons l'évolution de la probabilité de faute sur 50.000 injections de fautes, de manière à visualiser la vitesse de convergence de chaque technique d'optimisation (rapide ou lente). Le tableau 6.3 présente la probabilité de faute des meilleures configurations trouvées avec chaque technique. Avant d'aborder les résultats de SMAC et SHA, notez que les résultats de la recherche aléatoire peuvent fortement varier et même décroitre, en particulier pour µC-M3 et µC-M4. La recherche aléatoire nécessite plus d'injections de faute (>50.000) pour se stabiliser.

Pour chaque microcontrôleur, SMAC est significativement plus rapide que n'importe quelle technique d'optimisation pour identifier la configuration avec la probabilité de faute la plus élevée. En particulier, en moins de 10.000 injections de fautes, SMAC identifie systématiquement des configurations avec des probabilités de fautes plus élevées que GA, SHA et RS. Par conséquent, SMAC peut être utilisé pour optimiser un équipement d'injection de faute plus rapidement qu'avec les techniques d'optimisation traditionnelles, et donc, économiser du temps précieux pendant les évaluations de sécurité.

D'autre part, SHA converge lentement vers la meilleure configuration. Cependant, après les 50.000 injections de fautes, SHA trouve la configuration avec la probabilité de faute la plus élevée pour µC-M0 et µC-M3. Cela s'explique du fait de la conception de SHA. En effet SHA alloue entièrement le budget fixé T (le nombre d'injections de faute), et n'enlève que progressivement les pires configurations à chaque réduction de moitié, ce qui explique la vitesse de convergente lente par rapport aux autres techniques d'optimisation.

Néanmoins, nous trouvons que SHA évalue trop de configurations avec des probabilités de fautes faible, en particulier lors des premiers tours et sur le microcontrôleur µC-M0. Nous pensons que la procédure additionnelle proposée pour SMAC, décrite dans à la sous-section 6.4.3 peut aider SHA à sélectionner les configurations initiales $\vec{\Theta}_0$, de manière à réduire le temps passé sur des configurations peu performantes. Mais cela ne permettra sans doute pas de résoudre le problème déjà évoqué à la sous-section 6.4.2 à savoir, lorsque à l'itération k, aucune configuration ne permet de fauter le microcontrôleur, la sélection des configurations pour l'itération suivante $k+1$ se fera aléatoirement, sans consulter les résultats de l'itération $k-1$. Pour résumer, le véritable problème du SHA, c'est qu'il ne tient pas compte des résultats précédents. C'est pourquoi SMAC semble plus pertinent pour notre problème d'optimisation.

Bien que nous n'avons pas évalué SMAC ou SHA avec d'autres techniques d'injection de fautes, nous pensons que ces techniques d'optimisation devraient être adaptables pour optimiser des équipements d'injection EM et Laser. Pour terminer, au vu des résultats obtenus, SMAC est plus efficace que GS, SHA et RS, particulièrement pour optimiser rapidement un équipement d'injection de faute pour un microcontrôleur donné. C'est pourquoi, dans la section suivante, nous verrons que SMAC peut être utilisé pour exploiter des vulnérabilités plus rapidement qu'avec GA.

6.6 SMAC pour contourner des mécanismes de protection de code

Dans cette section, nous appliquons notre méthodologie en boîte noire avec SMAC dans le but de contourner un mécanisme de protection de code d'un microcontrôleur 32-bit, en utilisant notre `GLTICH STATION`. L'attaque présentée est une attaque connue, permettant d'abaisser le niveau de sécurité de la cible, de manière à extraire l'application embarquée [BFP19]. Nous verrons que SMAC est meilleur que GA à identifier les meilleures configurations en un nombre limité d'injections de faute, et ainsi montrer que SMAC permet de gagner un temps conséquent lors d'évaluations de sécurité.

6.6.1 STM32F103RB

Le microcontrôleur 32-bit STM32F103RB est basé sur un cœur ARM Cortex-M3 cadencé à 24Mhz. Le microcontrôleur STM32F103RB offre des mécanismes de protection permettant de bloquer la lecture ou l'écriture en mémoire utilisateur avec le *bootloader*. En pratique, une fois le mécanisme de protection de lecture activé (*ReaDout Protection*, RDP), le *bootloader*

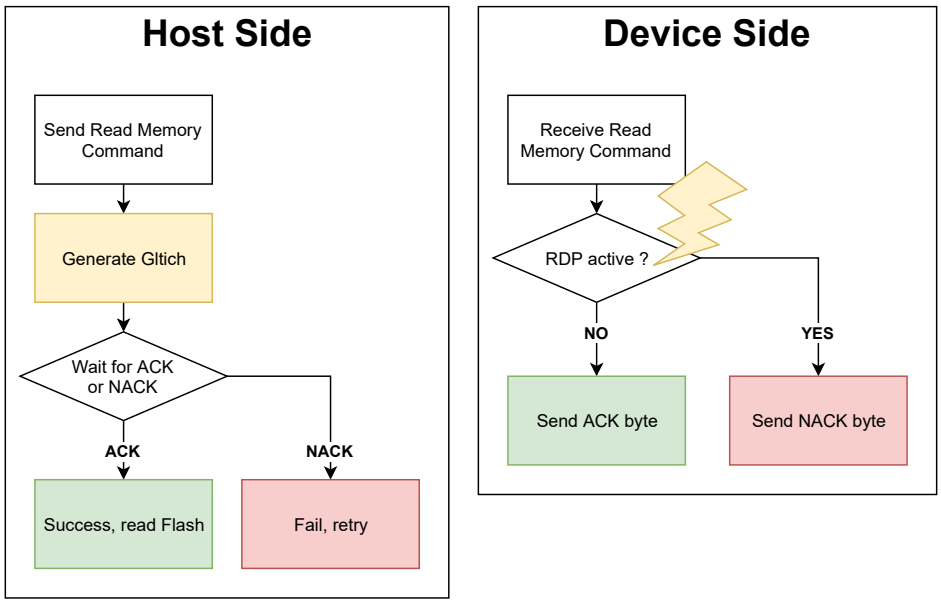


FIGURE 6.7 – Déroulement de l'attaque. Le but est de générer le glitch lors de la vérification du niveau de protection RDP pour forcer la lecture en mémoire Flash.

retourne une réponse négative (NACK) lorsque la commande *Read Memory* est émise, empêchant un utilisateur malveillant d'extraire le code en mémoire Flash. Pour désactiver RDP, la mémoire Flash doit être complétement effacée.

Attaque L'attaque connue pour contourner ce mécanisme consiste à injecter une faute pendant l'exécution de la commande *Read Memory* [BFP19]. En effet, lorsque le *bootloader* reçoit la commande *Read Memory*, il vérifie la valeur de l'octet contenant le niveau de protection RDP, et retourne la réponse ACK ou NACK, selon si RDP est désactivé ou activé, respectivement. En injectant une faute pendant l'étape de vérification de la valeur de l'octet RDP, un attaquant peut tromper le mécanisme de protection en lecture et récupérer le contenu du bloc mémoire sélectionné (figure 6.7).

Caractérisation Nous avons utilisé les techniques d'optimisation SMAC et GA pour identifier les meilleurs réglages pour fauter le STM32F103RB avec notre `GLTICH STATION`. Pour chaque technique d'optimisation, 6000 fautes ont été injectées (soit 24 générations pour le GA) pendant l'étape de caractérisation (<10min), avec les paramètres par défaut (tableau 6.2) et le test de caractérisation de faute IC (tableau 5.12). La figure 6.8 présente la probabilité de faute des meilleures configurations trouvées avec SMAC et GA en fonction du nombre d'injections de faute. On constate que SMAC est significativement plus rapide à identifier les meilleures configurations que GA. Non seulement SMAC converge plus rapidement que GA, mais SMAC identifie également des configurations avec des probabilités de fautes deux fois plus élevées qu'avec GA, comme présenté au tableau 6.4.

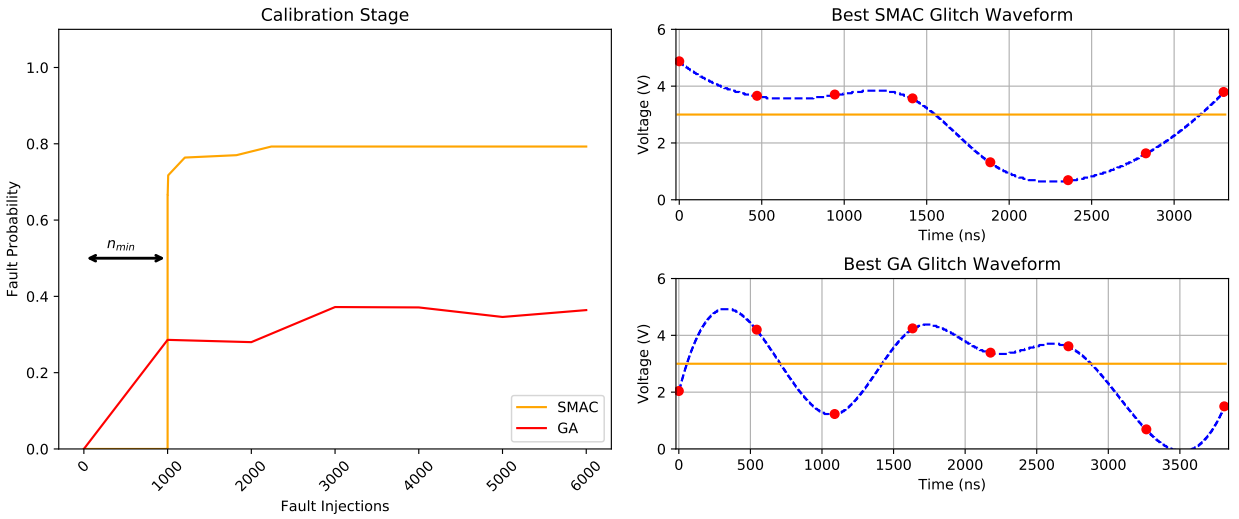


FIGURE 6.8 – Évolution de la probabilité de faute sur 6000 injections de fautes, en utilisant SMAC et GA, sur le STM32F103RB ; et les meilleures formes de *glitch* trouvées avec SMAC et GA pendant l'étape de caractérisation.

		Nombre d'injections de fautes	
		6000	12000
SMAC	Probabilité de faute Max	**0.79**	0.79
	Temps de caractérisation	15 min	30 min
	Temps d'exploitation	**<5 min**	<5 min
GA	Probabilité de faute Max	0.37	0.55
	Temps de caractérisation	15 min	30 min
	Temps d'exploitation	N/A	<5 min

TABLE 6.4 – Comparaison de la performance entre SMAC et GA sur le microcontrôleur STM32F103RB avec la `GLITCH STATION`.

Exploitation Nous vérifions que les configurations identifiées avec chacune des techniques d'optimisation testées permettent de contourner le mécanisme de sécurité RDP sur le microcontrôleur STM32F103RB. De plus, nous comparons le temps écoulé moyen pour réussir l'attaque en utilisant les différentes configurations trouvées avec SMAC et GA pour notre `GLITCH STATION`. Les temps d'exploitation sont détaillés au tableau 6.4. Tandis que l'attaque est facilement réalisable avec la configuration trouvée avec SMAC, *en moyenne en moins de 5 minutes*, aucune des configurations trouvées avec GA ne permettent de contourner le mécanisme de protection en lecture du STM32F103RB. Cela montre qu'avec seulement 6000 injections de fautes lors de l'étape caractérisation, GA sous-performe nettement SMAC pour cet exemple. La figure 6.9 présente les traces de l'oscilloscope au moment de l'injection de faute pour contourner le mécanisme RDP du STM32F103RB, avec la meilleure configuration

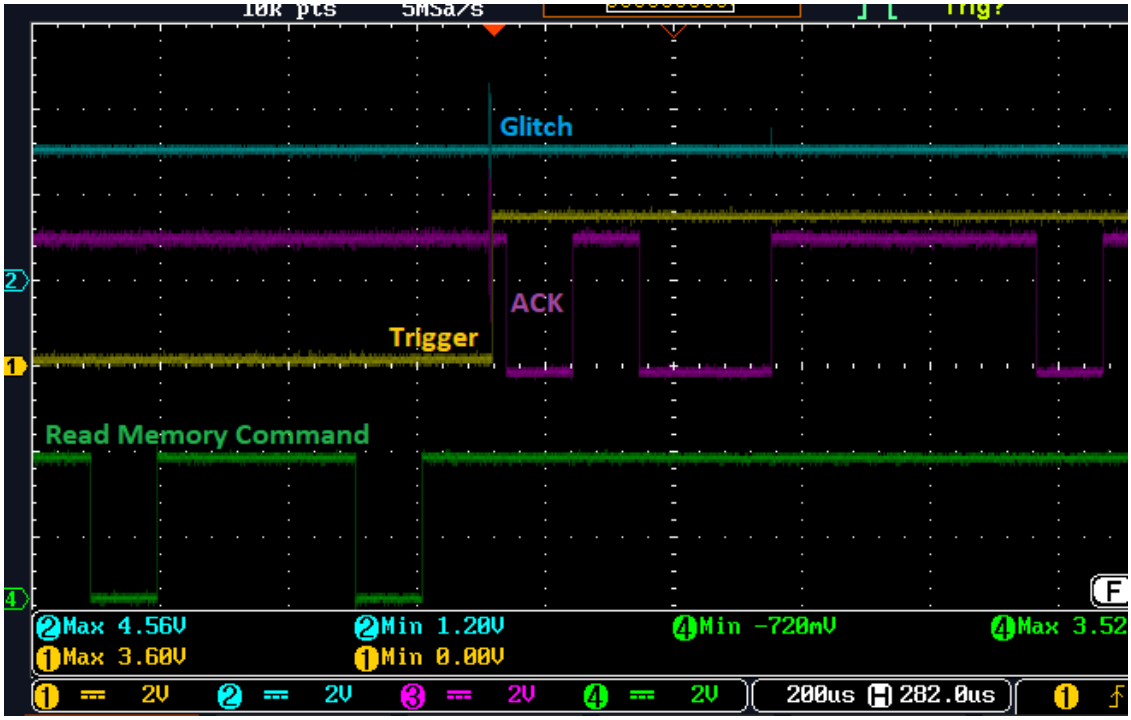


FIGURE 6.9 – Les traces de l'oscilloscope au moment de l'injection de faute pour contourner le mécanisme RDP du STM32F103RB.

identifiée avec SMAC.

Notez qu'avec un plus grand nombre d'injections de faute lors de l'étape de caractérisation, il est aussi possible de réussir l'attaque avec GA. Par exemple, avec deux fois plus d'injections de faute pendant l'étape de caractérisation (i.e. 12.000 au lieu de 6.000), GA identifie des configurations d'équipement permettant de fauter le STM32F103RB et de contourner le mécanisme RDP, comme présenté au tableau 6.4. Cependant, même avec deux fois plus d'injections de fautes, les configurations identifiées avec GA ont une probabilité de faute plus faible que SMAC.

6.7 Conclusion

Dans ce chapitre, nous avons proposé d'appliquer deux techniques d'optimisation récentes pour identifier les meilleurs paramètres de notre `GLITCH STATION`, sur plusieurs microcontrôleurs 32-bit différents. Bien que ces techniques ont déjà été utilisées avec succès en apprentissage automatique ou dans la résolution de problèmes combinatoires difficiles, c'est la première fois qu'elles sont utilisées pour optimiser l'injection de faute. L'optimisation bayésienne (SMAC) ou les algorithmes de bandit (SHA) permettent d'explorer rapidement l'espace des paramètres afin de maximiser les fautes exploitables sur différents microcontrôleurs.

Si SHA est un algorithme simple et facilement adaptable à notre problème d'optimisa-

tion, identifiant sous certaines conditions les meilleures configurations, c'est bien SMAC qui a retenu notre attention. Ce configurateur automatique permet de trouver les meilleurs paramètres d'équipement tout en étant bien plus rapide que les algorithmes métaheuristiques.

De plus, pour simplifier et accélérer l'identification et l'exploration, nous proposons de séparer l'optimisation en deux étapes, à savoir la *caractérisation* et l'*exploitation*. Tout d'abord, nous optimisons avec SMAC ou SHA les paramètres d'équipement indépendamment de l'application cible en utilisant un test de caractérisation de faute. Une fois les meilleures configurations identifiées, une recherche par quadrillage permet d'identifier les délais d'injection de fautes pour exploiter les vulnérabilités sur l'application cible.

En particulier, avec SMAC et notre approche, nous arrivons à contourner le mécanisme de protection en lecture implémenté dans le *bootloader* préchargé du microcontrôleur STM32F103RB, deux fois plus rapidement qu'avec un GA.

De plus, SMAC et SHA ont systématiquement identifié de meilleures configurations que les algorithmes métaheuristiques. Trouver plus facilement les paramètres d'équipement avec une probabilité de faute élevée va permettre d'améliorer notre méthodologie d'injection de fautes multiples (chapitre 4), en réduisant la difficulté de l'exploitation d'attaques multi-fautes complexes.

Par la suite, il pourrait être pertinent d'évaluer d'autres techniques d'optimisation alternatives telles que HyperBand ou BO-TPE. De plus, il serait intéressant d'évaluer les performances de SMAC et SHA avec d'autres techniques d'injection de fautes, telles que l'injection EM ou Laser. Enfin, il faudrait étudier les applications directes de SMAC ou SHA sur des microcontrôleurs sécurisés, comme l'identification de formes de *glitch* exotiques avec SMAC pouvant potentiellement contourner des contre-mesures matérielles comme des détecteurs.

Chapitre 7

Conclusion

Sommaire

7.1 Bilan

Synthèse Générale Les vulnérabilités à l'injection de faute sont complexes à étudier car, en plus des connaissances matérielles et logicielles requises pour analyser l'architecture du microcontrôleur ciblé et l'application embarquée dans celui-ci, il faut aussi maitriser l'art subtil de la calibration des paramètres d'équipement, avant de réussir à générer des fautes sur la cible. Dans ce contexte, l'objectif de cette thèse était de proposer de nouvelles méthodes, techniques et outils pour optimiser l'identification et l'exploitation de ces vulnérabilités, soit en comblant l'écart entre simulation et expérimentation, soit en utilisant de nouvelles approches pour identifier plus efficacement les paramètres induisant des fautes sur les microcontrôleurs. Si la volonté de mieux comprendre les fautes sur les systèmes embarqués nous a conduit à développer plusieurs solutions différentes, de nombreuses pistes de recherche pour de futurs travaux restent ouvertes. La suite de cette section résume les travaux développés dans ce manuscrit.

Dans un premier temps, au chapitre 2, nous avons présenté les différents moyens et techniques existants pour injecter des fautes ainsi que les attaques récentes sur des microcontrôleurs non sécurisés. Nous avons aussi dressé un état de l'art des techniques d'optimisation des paramètres d'équipement et du délai d'injection. En particulier, nous avons constaté l'écart entre les techniques utilisées en injection de faute et celles utilisées dans d'autres domaines comme l'apprentissage automatique, plus récentes et plus efficaces. Nous avons aussi présenté l'état de l'art de la caractérisation de faute, et en particulier des tests de caractérisation de fautes. Pour ces derniers, nous avons observé l'absence de métriques et de critères pour les comparer objectivement. Enfin, nous avons constaté que les vulnérabilités aux fautes multiples n'étaient pas prises en compte dans les méthodologies proposées récemment, et que

même si les modèles spécifiques à la cible sont de plus en plus utilisés pour simuler les fautes, la transition de la simulation vers l'expérimentation n'est pas abordée.

Comme l'identification et l'exploitation de vulnérabilités à l'injection de faute passent par l'usage d'outils performants, au chapitre 3 nous avons détaillé les outils développés et utilisés pendant cette thèse, `CELTIC` et la `GLITCH STATION`. `CELTIC` fut l'objet d'une refonte complète et d'un travail d'ingénierie important avec l'ajout en particulier du `multithreading` et de la précompilation du jeu d'instructions cible pour accélérer la simulation d'injection de faute. La `GLITCH STATION` est un nouvel outil *low cost* entièrement développé pendant cette thèse permettant de générer des *power glitches* très précis. À l'aide d'un convertisseur numérique-analogique, la `GLITCH STATION` peut ajuster finement la forme de la perturbation, permettant d'injecter des fautes sur une grande variété de microcontrôleurs.

Au chapitre 4, nous avons proposé une méthodologie visant à faciliter l'identification et l'exploitation de vulnérabilités à l'injection de fautes multiples. La simulation de faute est limitée uniquement aux modèles de faute spécifiques à la cible identifiés au préalable avec une caractérisation, à l'aide de `CELTIC`. Cela permet de limiter l'explosion combinatoire du nombre de chemins d'attaque avec le nombre de fautes indépendantes injectées, et de détecter uniquement des vulnérabilités réalistes, dont les paramètres d'équipement associés sont automatiquement générées pour faciliter l'exploitation de ces dernières. En moyenne, en se basant sur cinq expériences avec plusieurs cibles et différentes techniques d'injection, nous avons exploité 10 fois plus rapidement des vulnérabilités par injection de fautes multiples avec notre approche bout-en-bout qu'avec une caractérisation de faute classique. Nous avons aussi discuté des possibles limitations de notre approche et proposé plusieurs pistes pour l'améliorer.

En particulier, nos efforts se sont concentrés sur l'optimisation de l'étape de caractérisation, cruciale pour le reste de notre méthodologie. C'est pourquoi, au chapitre 5, nous avons formalisé deux propriétés essentielles pour les tests de caractérisation, la *propagation* et la *discrimination* des fautes, qui furent ensuite dérivées en plusieurs métriques de performance. À partir de ces métriques, nous avons étudié et comparé plusieurs tests de la littérature pour identifier les critères les plus importants selon les modèles de faute considérés. Finalement, à partir de ces recommandations, nous avons proposé un ensemble de tests optimaux qui furent utilisés avec la `GLITCH STATION` pour identifier rapidement les modèles de fautes les plus probables sur un Cortex-M4 32-bit.

Toujours pour améliorer la caractérisation de faute, au chapitre 6, nous avons proposé de nouvelles techniques d'optimisation pour accélérer l'identification des meilleurs paramètres d'équipement en fonction du microcontrôleur ciblé. Nous avons montré expérimentalement que les algorithmes SHA et SMAC permettent d'identifier plus rapidement les paramètres de la `GLTICH STATION` que les algorithmes métaheuristiques. En particulier, nous avons exploité une vulnérabilité connue sur un STM32F103RB avec SMAC 2 fois plus rapidement qu'avec un algorithme génétique. Aussi, au lieu d'optimiser l'équipement directement sur l'application ciblée, nous avons proposé une optimisation indirecte de l'équipement via des tests de caractérisation, ce qui permet déterminer les paramètres d'équipement indépendamment du délai d'injection, et ainsi réduire l'espace de recherche.

7.2 Perspectives

Les travaux conduits pendant cette thèse permettent d'ouvrir plusieurs axes de recherche afin d'améliorer la sécurité des systèmes embarqués. Dans les paragraphes suivants, nous détaillons plusieurs pistes potentielles que nous avons identifié, toujours dans le but faciliter la détection de vulnérabilités à l'injection de faute.

Combinaison des solutions proposées Tout d'abord, pour conclure ces travaux, et confirmer la pertinence de l'ensemble de tests optimaux proposé ainsi que des nouvelles techniques d'optimisation comme SMAC, il faudrait réinjecter ces améliorations dans notre méthodologie bout-en-bout présentée au chapitre 4. Il serait alors intéressant de comparer la couverture de fautes avec et sans ces améliorations, en particulier pour les microcontrôleurs moins sensibles aux fautes, et de vérifier si elles permettent d'améliorer l'inférence de modèle de faute.

Amélioration des modèles de fautes À plus long terme, il faudrait réfléchir sur l'implémentation de certaines pistes d'améliorations décrites à la fin du chapitre 4 comme l'extension de la base de donnée de modèles de faute avec des combinaisons de ces derniers dans le but de former de nouveaux modèles plus complexes pour couvrir plus de fautes, similairement aux travaux de Alshaer et al. [ACD+21]. En moyenne, notre approche permet de couvrir 76% des fautes, et nous pensons que l'utilisation de modèles plus complexes permettrait de se rapprocher de la couverture idéale de 100%. De plus, comme les modèles de faute sont aussi essentiels lors du développement de contremesures logicielles [Mor14], il est donc capital de continuer à améliorer la précision de ces derniers.

Émulation du pipeline avec `CELTIC` Il serait aussi intéressant d'améliorer le niveau de fidélité de `CELTIC`, en intégrant l'émulation complète du *pipeline*, similairement aux travaux de Yuce [Yuc18] qui utilise un simulateur de faute basé sur l'émulateur modulaire open source `GEM5` [BBB+11] pour émuler la microarchitecture ciblée. De cette manière, il est possible de suivre très précisément la propagation de fautes dans le système et de gagner en fidélité sur les vulnérabilités identifiées. Cependant, cela se ferait au détriment de la rapidité de l'outil, ce qui peut s'avérer être contre productif pour la simulation d'injection de fautes multiples.

Microcontrôleurs sécurisés L'autre point à intéressant à aborder dans de futurs travaux serait d'utiliser la `GLITCH STATION` avec SMAC pour évaluer un microcontrôleur sécurisé, dans le but d'analyser la robustesse de contre-mesures matérielles. Cela permettrait de vérifier s'il est possible de contourner les détecteurs de *power glitches* [VK19] en ajustant très précisément la forme de la perturbation avec la `GLITCH STATION`. De manière générale, nous souhaitons prochainement évaluer de nouveaux microcontrôleurs avec SMAC en utilisant des techniques d'injection de faute différentes. En particulier, l'injection de faute par impulsions électromagnétiques, réputée pour être difficile à maitriser, fait partie des candidats prioritaires. Contrairement à une recherche par quadrillage, où les déplacement spatiaux se font à intervalles réguliers, sollicitant donc peu les moteurs pas à pas, SMAC explore l'espace d'une

manière plus dynamique, ce qui exerce plus de pression sur ces derniers, pouvant conduire, dans certains cas, à des secousses si le banc est mal calibré (similairement aux imprimantes 3D). Cependant, mis à part ce problème d'ingénierie, SMAC devrait sans doute pouvoir optimiser la recherche des meilleurs paramètres d'équipement, y compris pour l'injection EM.

Injection de faute et Fuzzing Enfin à titre personnel, j'ai un intérêt tout particulier pour la sécurisation des systèmes embarqués, et plus précisément, des *bootloaders*. Le *bootloader* est une application particulièrement critique pour la sécurité du microcontrôleur et nous avons vu au chapitre 2 qu'il existe de nombreuses vulnérabilités connues sur plusieurs références utilisées encore aujourd'hui. En parallèle, le *fuzzing*, qui est une technique consistant à tester un grand nombre de données aléatoires dans un système, dans le but d'identifier des vulnérabilités, et de plus en plus utilisé en embarqué [CGS+20]. Cependant, l'utilisation du *fuzzing* en combinaison avec de l'injection de faute n'est à ma connaissance, pas référencée dans la littérature, et pourtant pourrait permettre d'identifier de nouveaux chemins d'attaque exotiques. C'est pourquoi, mes futurs travaux porteront vraisemblablement sur le développement d'un outil combinant *fuzzing* et injections de faute, tout en se basant sur les résultats obtenus pendant cette thèse pour optimiser l'identification et l'exploitation de vulnérabilités à l'injection de faute.

Annexes

```python
# On initialise un nouveau Project avec le fichier de description d'architecture GISL
# et le binaire à évaluer HEX. Le Project contient les propriétés de l'architecture ciblée
# et permet de créer divers objets, via la factory, comme l'état initial entry_state,
# ou encore le simulateur sim_engine.
prj = pyceltic.Project(GISL,HEX)

# Justemment avec la factory, on initialise l'état initial entry_state, et on ajoute
# des watch variables permettant de suivre facilement la valeur de variables globales
# en mémoire.
entry = prj.factory.entry_state()
entry.variables.add(pyceltic.MemoryView("g_countermeasure", 0x2000001c, 1))
entry.variables.add(pyceltic.MemoryView("g_authenticated", 0x2000001e, 1))

# Toujours avec la factory, on instancie un sim_engine pour exécuter le binaire jusqu'à
# la fonction VerifyPin à l'adresse 0x080002d4. On sauvegarde l'état courant avec clone().
engine = prj.factory.sim_engine(entry)
engine.step_until(0x080002d4)
verifypin_entry = engine.state.clone()

# La fonction end_procedure définit la fin de l'exécution. Elle prend en entrée un état state et
# retourne en sortie un booléen.
def end_procedure(state):
   return state.pc == 0x08000264

# La fonction timeout permet de signaler à CELTIC de génère une intérruption pour cause de boucle
# infinie. Elle prend en entrée un état state et retourne en sortie un booléen.
def timeout(state):
   return state.cycle > 500

# La fonction oracle permet de définir les conditions pour réussir une attaque. Ici, le but est
# d'être authentifié sans déclencher une contre-mesure. Elle prend en entrée un état state et
# retourne en sortie un booléen.
def oracle(state):
   return state.variables["g_authenticated"] == 0xaa and state.variables["g_countermeasure"] != 1

# On instancie un modèle de saut 16InstrSkipModel.
m1 = pyceltic.InstrSkipModel([16], prj.properties)

# Encore avec la factory, on instancie le simulateur d'injection de faute attack_engine. Il faut
# renseigner l'état intial à partir duquel injecter des fautes state, la liste de modèles à simuler
# models, le nombe de faute multiple à injecter n, et les fonctions end_procedure, oracle et timeout.
attack_engine = prj.factory.attack_engine(state=verifypin_entry,
                                          models=[m1],
                                          n=1,
                                          end_procedure=end_procedure,
                                          oracle=oracle,
                                          timeout=timeout)

# Enfin on peut lancer la simulation d'injection de faute avec run().
attack_res = attack_engine.run()
```

FIGURE A.I – Exemple de script `Python` d'utilisation de `CELTIC`.

```python
# On identifie les meilleurs paramètres d'équipement best avec la technique d'optimisation smac
best = smac.optimize()

# On récupère les points x0...x7 et la durée duration optimisés avec SMAC
x0 = best["x0"]
x1 = best["x1"]
x2 = best["x2"]
x3 = best["x3"]
x4 = best["x4"]
x5 = best["x5"]
x6 = best["x6"]
x7 = best["x7"]
points = [x0,x1,x2,x3,x4,x5,x6,x7]
duration = best["size"]

# On génère la forme de glitch waveform avec une interpolation entre les n_points points x0...x7
# La forme est tronquée (lower_bound, upper_bound) pour ne pas griller le circuit.
n_points = 8
lower_bound = 80
upper_bound = 180
x = np.linspace(0, n_points, num=n_points, endpoint=True)
f = interpolate.interp1d(x, points, kind="cubic")
x_ = np.linspace(0, n_points, num=duration, endpoint=True)
waveform = [min(max(int(i), lower_bound), upper_bound) for i in f(x_)]

try:
   # On envoie à la GLITCHUNIT la forme waveforme et la durée duration
   station.config_DMA(waveform, duration)
   # On configure le délai d'injection delay...
   station.config_delay(delay)
   # ... et la référence pour le délai d'injection, ici c'est le port d'entrées-sorties
   # gs.GlitchUnitTrigger.GPIO_PIN_11.
   station.launch(gs.GlitchUnitTrigger.GPIO_PIN_11)

   # Enfin on démarre la cible avec normal_boot()
   station.target.normal_boot()

   # On exécute le scénario d'attaque

   ...

   # Lorsque le scénario est terminé, on exécute une routine pour désactiver la GLITCHUNIT terminate()
   # et on éteint la cible avec poweroff().
   station.terminate()
   station.target.poweroff()
except gs.MCUBootloaderException as mbe:
   # Si une exception est capturée, on reset la GLITCHUNIT avec system_reset()
   station.system_reset()
```

FIGURE A.II – Exemple de script `Python` d'utilisation de la `GLITCHSTATION`.

	Test de Caractérisation de fautes T1				Test de Caractérisation de fautes T2			

I	`adds r0, #2`
	`adds r1, #3`
	`adds r2, #5`
	`adds r3, #7`
	`adds r4, #11`
	`adds r5, #13`
	`adds r6, #17`
	`adds r7, #19`
	} Repeat n times
					`mov r0, r0`
					`mov r1, r1`
					`mov r2, r2`
					`mov r3, r3`
					`mov r4, r4`
					`mov r5, r5`
					`mov r6, r6`
					`mov r7, r7`
					} Repeat n times			

V	R0	0x00000000	R1	0x11111111	R0	0x00000000	R1	0x11111111
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE A.I – Tests de caractérisation de fautes T1 and T2

	Test de Caractérisation de fautes T3				Test de Caractérisation de fautes T4			

I	`adds r0, #2`
	`adds r1, #3`
	`adds r2, #5`
	`adds r3, #7`
	`adds r4, #11`
	`adds r5, #13`
	`adds r6, #17`
	`adds r7, #19`
					`adds r0, #2`
					`adds r1, #3`
					`adds r2, #5`
					`adds r3, #7`
					`adds r4, #11`
					`adds r5, #13`
					`adds r6, #17`
					`adds r7, #19`
					} Repeat n times			

V	R0	0x00000000	R1	0x11111111	R0	0x00000000	R1	0x00000000
	R2	0x22222222	R3	0x33333333	R2	0x00000000	R3	0x00000000
	R4	0x44444444	R5	0x55555555	R4	0x00000000	R5	0x00000000
	R6	0x66666666	R7	0x77777777	R6	0x00000000	R7	0x00000000

TABLE A.II – Tests de caractérisation de fautes T3 and T4

	Test de Caractérisation de fautes T5				Test de Caractérisation de fautes T6			

I	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	} Repeat n times
					`ldr r0, =array ; [0,1,2,3,...]`
					`ldr r1, [r0]`
					`ldr r2, [r0, #4]`
					`ldr r3, [r0, #8]`
					`ldr r4, [r0, #12]`
					`ldr r5, [r0, #16]`
					`ldr r6, [r0, #20]`
					`ldr r7, [r0, #24]`			

V	R0	0x00000000	R1	0x11111111	R0	0x20005000	R1	0x11111111
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE A.III – Tests de caractérisation de fautes T5 and T6

	Test de Caractérisation de fautes T7				Test de Caractérisation de fautes T8			

I	`sub r7,r5`
	`add r8,r4`
	`subs r0, #0xff`
	`...`
	`lsl r1, r5, #8`
	`mov r2,r6`
	} n standard data-processing instructions
					`ldr r0, =array ; [r1,r2,...]`
					`str r1, [r0]`
					`ldr r1, [r0]`
					`...`
					`str r7, [r0, #24]`
					`ldr r7, [r0, #24]`
					} Repeat n times			

V	R0	0x00000000	R1	0x11111111	R0	0x20002000	R1	0x11111111
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE A.IV – Tests de caractérisation de fautes T7 and T8

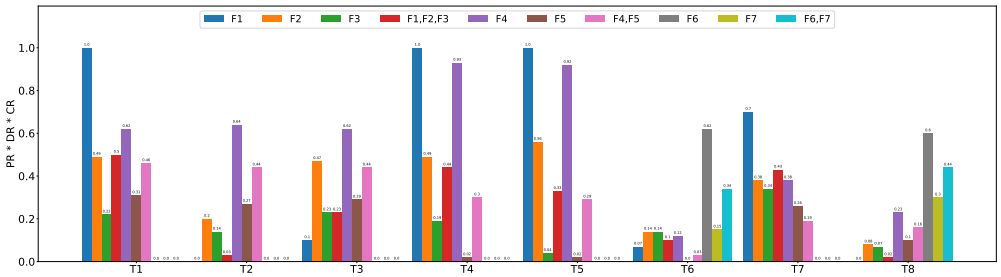


FIGURE A.III – Comparaison de $PR * DR * CR$ avec les tests de caractérisation de fautes du tableau 5.2 en fonction des modèles du tableau 5.1.

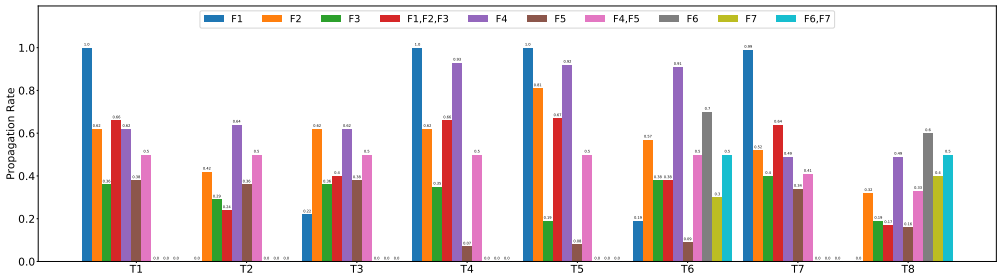


FIGURE A.IV – Comparaison de PR avec les tests de caractérisation de fautes du tableau 5.2 en fonction des modèles du tableau 5.1.

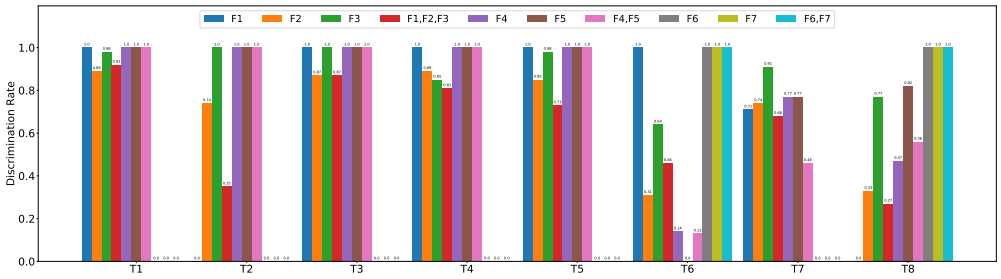


FIGURE A.V – Comparaison de DR avec les tests de caractérisation de fautes du tableau 5.2 en fonction des modèles du tableau 3.1.

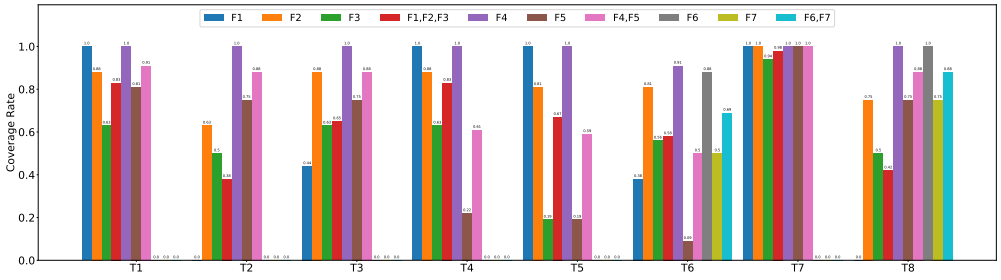


FIGURE A.VI – Comparaison de CR avec les tests de caractérisation de fautes du tableau 5.2 en fonction des modèles du tableau 5.1.

Instruction	Description	Exemple	
		Opcode	Mnémonique
ADC	Add with carry	0xEB410503	adc.w r5, r1, r3
ADD	Add	0xEB010203	add.w r2, r1, r3
ADR	Form PC-relative address	0xA301	adr r3, #4
AND	Bitwise AND	0xF402497F	and sb, r2, #0xff00
BIC	Bitwise bit clear	0xF02100AB	bic r0, r1, #0xab
EOR	Bitwise exclusive OR	0xF09B3718	eors r7, fp, #0x18181818
MOV	Copies operand to destination	0x0008	movs r0, r1
MVN	Bitwise NOT	0xEA6F0807	mvn.w r8, r7
ORN	Bitwise OR NOT	0xEA6B173E	orn r7, fp, lr, ror #4
ORR	Bitwise OR	0xEA400205	orr.w r2, r0, r5
RSB	Reverse subtract	0xF5C464A0	rsb.w r4, r4, #0x500
SBC	Subtract with carry	0x418B	sbcs r3, r1
SUB	Subtract	0xF1B608F0	subs.w r8, r6, #0xf0
ASR	Arithmetic shift right	0xFA48F709	asr.w r7, r8, sb
LSL	Logical shift left	0x4091	lsls r1, r2
LSR	Logical shift right	0xFA25F406	lsr.w r4, r5, r6
ROR	Rotate right	0xFA65F406	ror.w r4, r5, r6

TABLE A.V – Les 17 instructions ARMv7-M considérées (16-bit et 32-bit).

Bibliographie

[AAA+90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation : A methodology and some applications. *IEEE Transactions on software engineering*, 16(2) :166–182, 1990. 6

[ABC+17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. Nanofocused x-ray beam to reprogram secure circuits. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 175–188. Springer, 2017. 11

[ACD+21] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. Microarchitecture-aware fault models : Experimental evidence and cross-layer inference methodology. In *2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2021. 30, 31, 51, 75, 123

[AGH19] Karim Abdellatif, Charles Guillemet, and Olivier Hériveaux. Unfixable seed extraction on trezor - a practical and reliable attack. `https://donjon.ledger.com/Unfixable-Key-Extraction-Attack-on-Trezor`, 2019. 13

[AH20] Karim M Abdellatif and Olivier Hériveaux. Silicontoaster : a cheap and programmable em injector for extracting secrets. In *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 35–40. IEEE, 2020. 10, 11

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance-a cautionary note. In *Proceedings of the second Usenix workshop on electronic commerce*, volume 2, pages 1–11, 1996. 7, 8

[AK97] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, pages 125–136. Springer, 1997. 8, 42

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. *Department of Computing Science Technical Report Series*, 2001. 6

[ALSU20] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers : principles, techniques and tools*. 2020. 37

[Azi19] Maryam Aziz. *On Multi-Armed Bandits Theory and Applications*. PhD thesis, Ph. D. Thesis, Northeastern University, Boston, MA, USA, 2019. 104, 111

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012. 16, 17

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2) :1–7, 2011. 123

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011. 111, 112

[BBC+14] Maël Berthier, Julien Bringer, Hervé Chabanne, Thanh-Ha Le, Lionel Rivière, and Victor Servant. Idea : embedded fault injection simulator on smartcard. In *International Symposium on Engineering Secure Software and Systems*, pages 222–229. Springer, 2014. 31

[BCZ+97] Donald A Berry, Robert W Chen, Alan Zame, David C Heath, and Larry A Shepp. Bandit problems with infinitely many arms. *The Annals of Statistics*, pages 2103–2116, 1997. 104

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997. 7

[BDSG+14] Johannes Blömer, Ricardo Gomes Da Silva, Peter Günther, Juliane Krämer, and Jean-Pierre Seifert. A practical second-order fault attack against a real-world pairing implementation. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 123–136. IEEE, 2014. 8

[Bel61] Richard E Bellman. *Adaptive control processes*. Princeton university press, 1861. 16

[BEMP20] Etienne Boespflug, Cristian Ene, Laurent Mounier, and Marie-Laure Potet. Countermeasures optimization in multiple fault-injection context. In *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 26–34. IEEE, 2020. 30, 32

[BF85] Donald A Berry and Bert Fristedt. Bandit problems : sequential allocation of experiments (monographs on statistics and applied probability). *London : Chapman and Hall*, 5(71-87) :7–7, 1985. 103

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch : optimizing voltage fault injection attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 199–224, 2019. 11, 12, 15, 17, 20, 42, 43, 46, 99, 116, 117

[BGV11] Josep Balasch, Benedikt Gierlichs, and Ingrid 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*, pages 105–114. IEEE, 2011. 9, 25, 26, 83

[BH15] Jakub Breier and Wei He. Multiple fault attack on present with a hardware trojan implementation in fpga. In *2015 international workshop on secure internet of things (SIot)*, pages 58–64. IEEE, 2015. 8

[BJ15] Jakub Breier and Dirmanto Jap. Testing feasibility of back-side laser fault injection on a microcontroller. In *Proceedings of the WESS'15 : Workshop on Embedded Systems Security*, pages 1–6, 2015. 11

[BK06] Johannes Blömer and Volker Krummel. Fault based collision attacks on aes. In *International Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 106–120. Springer, 2006. 7

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1) :5–32, 2001. 107

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997. 7

[BS02] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1) :3–52, 2002. 110, 113

[BYC+13] James Bergstra, Dan Yamins, David D Cox, et al. Hyperopt : A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, volume 13, page 20. Citeseer, 2013. 112

[CBD+15] Clément Champeix, Nicolas Borrel, Jean-Max Dutertre, Bruno Robisson, Mathieu Lisart, and Alexandre Sarafianos. Seu sensitivity and modeling using pico-second pulsed laser stimulation of a d flip-flop in 40 nm cmos technology. In *2015 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFTS)*, pages 177–182. IEEE, 2015. 29

[Ces16] Silvio Cesare. Defeating firmware copy protection using glitching. `https://www.youtube.com/watch?v=UGlm_I-RI-U`, 2016. 12

[CGS+20] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias

Payer. Halucinator : Firmware re-hosting through abstraction layer emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1201–1218, 2020. 124

[CJ05] Mathieu Ciet and Marc Joye. Practical fault countermeasures for chinese remaindering based rsa. In *Workshop on Fault Diagnosis and Tolerance in Cryptography–FDTC*, volume 5, pages 124–132. Citeseer, 2005. 8

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012. 31

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 181–194. Springer, 2007. 7

[CLMFT14] Franck Courbon, Philippe Loubet-Moundi, Jacques JA Fournier, and Assia Tria. Increasing the efficiency of laser fault injections using fast gate level reverse engineering. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 60–63. IEEE, 2014. 19, 20, 98

[CMD+18] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory : Instructions corruption on a 32-bit microcontroller. *IACR Cryptology ePrint Archive*, 2018 :1042, 2018. 22, 25, 26, 29, 40, 49, 54, 83, 84, 88

[CP95] Jeffrey A Clark and Dhiraj K Pradhan. Fault injection : A method for validating computer-system dependability. *Computer*, 28(6) :47–56, 1995. 6

[CPB+13] Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. Glitch it if you can : parameter search strategies for successful fault injection. In *International Conference on Smart Card Research and Advanced Applications*, pages 236–252. Springer, 2013. 15, 17, 20, 99, 100

[CPHR21] Ludovic Claudepierre, Pierre-Yves Péneau, Damien Hardy, and Erven Rohou. Traitor : A low-cost evaluation platform for multifault injection. In *ASIACCS 2021-16th ACM ASIA Conference on Computer and Communications Security*, pages 1–6. ACM, 2021. 11

[DBC+18] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hely, Regis Leveugle, Paolo Maistri, et al. Laser fault injection at the cmos 28 nm technology node : an analysis of the fault model. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–6. IEEE, 2018. 28

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012. 10

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. Sifa : exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 547–572, 2018. 8

[DLM20] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Modeling and simulating electromagnetic fault injection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(4) :680–693, 2020. 28

[DPdC+15] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks : Filling the gap using fault model inference. In *International conference on smart card research and advanced applications*, pages 107–124. Springer, 2015. 30, 40, 50, 51, 52, 54, 55, 83

[DPP+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. Fissc : A fault injection and simulation secure collection. In *International Conference on Computer Safety, Reliability, and Security*, pages 3–11. Springer, 2016. 41, 60

[DRPR19] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental analysis of the laser-induced instruction skip fault model. In *Nordic Conference on Secure IT Systems*, pages 221–237. Springer, 2019. 10, 25, 26, 29, 40, 49, 54, 83, 87

[Dur16] Louis Dureuil. *Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute*. PhD thesis, 2016. 29, 34, 35, 58, 73, 91

[FJLT13] Thomas Fuhr, Eliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on aes with faulty ciphertexts only. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–118. IEEE, 2013. 7

[Fle11] Bill Fleming. Microcontroller units in automobiles [automotive electronics]. *IEEE Vehicular Technology Magazine*, 6(3) :4–8, 2011. 1

[Ger17] Chris Gerlinsky. Breaking code read protection on the nxp lpc-family microcontrollers. `https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-Breaking_CRP_on_NXP_LPC_Microcontrollers_slides.pdf`, 2017. 12

[Gir05] Christophe Giraud. Fault resistant rsa implementation. *Fault Diagnosis and Tolerance in Cryptography*, pages 142–151, 2005. 8

[GT13] Anupriya Gogna and Akash Tayal. Metaheuristics : review and application. *Journal of Experimental & Theoretical Artificial Intelligence*, 25(4) :503–526, 2013. 17

[GWJL18] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. Bridging software-based and hardware-based fault injection vulnerability detection. 2018. 30, 50, 51

[Hat07] Simon Hatthon. Fitness proportionate selection. `https://commons.wikimedia.org/wiki/File:Fitness_proportionate_selection_example.png`, 2007. 18

[HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011. 3, 105, 106, 107, 109

[HHLBM09] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin P Murphy. An experimental investigation of model-based parameter optimisation : Spo and beyond. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 271–278, 2009. 107

[HLB19] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. Formally verified software countermeasures for control-flow integrity of smart card c code. *Computers & Security*, 85 :202–224, 2019. 48, 60, 66

[HS13] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013. 9, 10

[Hum14] Tim Hummel. Exploring effects of electromagnetic fault injection on a 32-bit high speed embedded device microprocessor. Master's thesis, University of Twente, 2014. 23, 24, 25, 26, 29

[Joi20] Joint Interpretation Working Group. Application of attack potential to smartcards and similar devices. 2020. 30, 48

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them : An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3) :361–372, 2014. 5

[Kei17] Keil. Using cortex-m3/m4/m7 fault exceptions. `https://www.keil.com/appnotes/files/apnt209.pdf`, 2017. 23

[Ken00] Michael Peter Kennedy. On the robustness of r-2r ladder dacs. *IEEE Transactions on Circuits and Systems I : Fundamental Theory and Applications*, 47(2) :109–116, 2000. 45

[KH14] Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 8–17. IEEE, 2014. 25, 26, 29, 40, 83

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976. 20

[KKS13] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246. PMLR, 2013. 103, 104

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Fault attacks for crt based rsa : New attacks, new results, and new countermeasures. In *IFIP International Workshop on Information Security Theory and Practices*, pages 215–228. Springer, 2007. 8, 31

[Kra20] Kraken. Kraken identifies critical flaw in trezor hardware wallets. `https://blog.kraken.com/post/3662/kraken-identifies-critical-flaw-in-trezor-hardware-wallets`, 2020. 13

[LA04] Chris Lattner and Vikram Adve. Llvm : A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. 36, 37

[Lap92] Jean-Claude Laprie. Dependability : Basic concepts and terminology. In *Dependability : Basic Concepts and Terminology*, pages 3–245. Springer, 1992. 6

[LBD+18] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the importance of analysing microarchitecture for accurate software fault models. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 561–564. IEEE, 2018. 28

[LBD+19] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a risc-v processor. *Microprocessors and Microsystems*, 71 :102862, 2019. 49, 69

[LDPPB21] J Laurent, C Deleuze, F Pebay-Peyroula, and V Beroulle. Bridging the gap between rtl and software fault injection. *ACM Journal of Emerging Technologies in Computing System*, 17(3) :1–24, 2021. 31, 50, 51

[LEF+17] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. Smac v3 : Algorithm configuration in python. `https://github.com/automl/SMAC3`, 2017. 113

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card c codes. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2014. 30

[LJD+17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband : A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1) :6765–6816, 2017. 105, 110, 111

[LL12] Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A : Statistical Mechanics and its Applications*, 391(6) :2193–2196, 2012. 59, 113

[Lu19] Yifan Lu. Injecting software vulnerabilities with voltage glitching. *arXiv preprint arXiv :1903.08102*, 2019. 13

[M+11] Wes McKinney et al. pandas : a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing 14*, 2011. 36

[MAM17] Maxime Madau, Michel Agoyan, and Philippe Maurine. An em fault injection susceptibility criterion and its application to the localization of hotspots. In *International Conference on Smart Card Research and Advanced Applications*, pages 180–195. Springer, 2017. 20, 99

[MBB16] Fabien Majéric, Eric Bourbao, and Lilian Bossuet. Electromagnetic security tests for soc. In *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 265–268. IEEE, 2016. 10

[MDH+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013. 10, 24, 25, 26, 83

[MOG+20] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt : Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020. 5

[Mor14] Nicolas Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Paris 6, 2014. 123

[MSPB19] Antun Maldini, Niels Samwel, Stjepan Picek, and Lejla Batina. Optimizing electromagnetic fault injection with genetic algorithms. In *Automated Methods in Cryptographic Fault Analysis*, pages 281–300. Springer, 2019. 14, 17, 20, 99

[Muk11] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011. 6

[OC14] Colin O'Flynn and Zhizhang David Chen. Chipwhisperer : An open-source platform for hardware embedded security research. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 243–260. Springer, 2014. 42, 43

[OGST+14] Sébastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, J-M Dutertre, and Philippe Maurine. Evidence of a larger em-induced fault model. In *International Conference on Smart Card Research and Advanced Applications*, pages 245–259. Springer, 2014. 10

[Osw16] David Oswald. Generic implementation analysis toolkit. `https://sourceforge.net/projects/giant`, 2016. 42, 43

[OT17] Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017. 12

[PBBJ15] Stjepan Picek, Lejla Batina, Pieter Buzing, and Domagoj Jakobovic. Fault injection with a new flavor : Memetic algorithms make a difference. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 159–173. Springer, 2015. 14, 17, 20, 99

[PBJC14] Stjepan Picek, Lejla Batina, Domagoj Jakobović, and Rafael Boix Carpi. Evolving genetic algorithms for fault injection attacks. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1106–1111. IEEE, 2014. 17, 20, 99

[PCHR20] Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy, and Erven Rohou. Nop-oriented programming : Should we care ? In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 694–703. IEEE, 2020. 49, 63, 95

[PHM+19] Julien Proy, Karine Heydemann, Fabien Majéric, Albert Cohen, and Alexandre Berzati. Studying em pulse effects on superscalar microarchitectures at isa level. *arXiv preprint arXiv :1903.02623*, 2019. 22, 25, 26, 29, 54, 61, 64, 83, 87

[PMPD14] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart : A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014. 30, 31

[PTH+15] Athanasios Papadimitriou, Marios Tampas, David Hély, Vincent Beroulle, Paolo Maistri, and Régis Leveugle. Validation of rtl laser fault injection model with respect to layout information. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 78–81. IEEE, 2015. 28, 29

[RDN19] Thomas Roth, Josh Datko, and Dmitry Nedospasov. Chip.fail. `https://chip.fail/chipfail.pdf`, 2019. Black Hat USA. 12, 15

[Ris17] Riscure. Vc glitcher. `https://www.riscure.com/uploads/2017/07/datasheet_vcglitcher.pdf`, 2017. 42, 43

[RNR+15] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015. 25, 26, 29, 40, 49, 54, 65, 83

[RPL+14] Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne, and Maxime Puys. Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In *International Symposium on Foundations and Practice of Security*, pages 92–111. Springer, 2014. 31, 50, 51

[RSDT13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in sram memory cells. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 89–98. IEEE, 2013. 28

[SA02] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002. 10

[Seg06] Segger. J-link edu. `https://www.segger.com/products/debug-probes/j-link/models/j-link-edu/`, 2006. 24

[SFR+15] Falk Schellenberg, Markus Finkeldey, Bastian Richter, Maximilian Schäpers, Nils Gerhardt, Martin Hofmann, and Christof Paar. On the complexity reduction of laser fault injection campaigns using obic measurements. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 14–27. IEEE, 2015. 19, 20, 98

[SH07] Jörn-Marc Schmidt and Michael Hutter. *Optical and em fault-attacks on crt-based rsa : Concrete results.* na, 2007. 10

[SHS16] Bodo Selmke, Johann Heyszl, and Georg Sigl. Attack on a dfa protected aes by simultaneous laser fault injections. In *2016 Workshop on Fault Diagnosis*

and Tolerance in Cryptography (FDTC), pages 36–46. IEEE, 2016. 8, 11, 61, 64

[Sko05] Sergei Petrovich Skorobogatov. Semi-invasive attacks : a new approach to hardware security analysis. 2005. 19

[Sko09] Sergei Skorobogatov. Local heating attacks on flash memory devices. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 1–6. IEEE, 2009. 9

[Sko10] Sergei Skorobogatov. Optical fault masking attacks. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 23–29. IEEE, 2010. 11

[Sli19] Aleksandrs Slivkins. Introduction to multi-armed bandits. *arXiv preprint arXiv :1904.07272*, 2019. 103

[Sty13] Erich Styger. Using the freedom board as swd programmer. `https://mcuoneclipse.com/2013/04/21/using-the-freedom-board-as-jtag-programmer/`, 2013. 24

[TBC19] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. Fault injection characterization on modern cpus. In *IFIP International Conference on Information Security Theory and Practice*, pages 123–138. Springer, 2019. 22, 25, 26, 29, 78, 83, 84, 88

[TBE+21] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. Electromagnetic fault injection against a complex cpu, toward new micro-architectural fault models. *Journal of Cryptographic Engineering*, pages 1–15, 2021. 10, 25, 26, 29

[TK10] Elena Trichina and Roman Korkikyan. Multi fault laser attacks on protected crt-rsa. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 75–86. IEEE, 2010. 8

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. {CLKSCREW} : exposing the perils of security-oblivious energy management. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1057–1074, 2017. 5

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling pc on arm using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016. 25, 26, 29, 42, 54, 78, 83, 87

[VdHOGT21] Jan Van den Herrewegen, David Oswald, Flavio D Garcia, and Qais Temeiza. Fill your boots : Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 56–81, 2021. 8, 12, 20, 31

[VK19] Ali Vosoughi and Selçuk Köse. Leveraging on-chip voltage regulators against fault injection attacks. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 15–20, 2019. 123

[VRD+00] Guido Van Rossum, Fred L Drake, et al. *Python reference manual*. iUniverse Indiana, 2000. 36

[WMP20] Vincent Werner, Laurent Maingault, and Marie-Laure Potet. An end-to-end approach for multi-fault attack vulnerability assessment. In *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 10–17. IEEE, 2020. 48

[WP83] Thomas W Williams and Kenneth P Parker. Design for testability—a survey. *Proceedings of the IEEE*, 71(1) :98–112, 1983. 6

[WRBBP20] Lichao Wu, Gerard Ribera, Noemie Beringuier-Boher, and Stjepan Picek. A fast characterization method for semi-invasive fault injection attacks. In *Cryptographers' Track at the RSA Conference*, pages 146–170. Springer, 2020. 19, 20, 99

[YS20] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms : Theory and practice. *Neurocomputing*, 415 :295–316, 2020. 19, 99, 104, 105

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software : Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2) :111–130, 2018. 27

[Yuc18] Bilgiday Yuce. *Fault attacks on embedded software : New directions in modeling, design, and mitigation*. PhD thesis, Virginia Tech, 2018. 123