Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle

Johan Laurent

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Nanoélectronique et Nanotechnologies**

Arrêté ministériel : 25 mai 2016

Présentée par

Johan LAURENT

Thèse dirigée par **Vincent BEROULLE, Professeur des universités, Grenoble INP - Esisar**, et
Co-encadrée par **Christophe DELEUZE, Maître de Conférences, Grenoble INP - Esisar** et **Florian PEBAY-PEYROULA, Ingénieur de Recherche, CEA-Leti**

préparée au sein du **Laboratoire de Conception et d'Intégration des Systèmes (LCIS)**
dans **l'École Doctorale EEATS**

Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle

Thèse soutenue publiquement le **19 novembre 2020**,
devant le jury composé de :

M. Bruno ROUZEYRE
Professeur à l'Université de Montpellier, Président
Mme. Karine HEYDEMANN
Maître de Conférences à Sorbonne Université, Rapportrice
M. Jean-Max DUTERTRE
Professeur à l'école des Mines de Saint-Etienne, Rapporteur
M. Lilian BOSSUET
Professeur à l'Université de Saint-Etienne, Examinateur
M. Vincent BEROULLE
Professeur à l'Université Grenoble Alpes, Directeur de thèse
M. Christophe DELEUZE
Maître de Conférences à l'Université Grenoble Alpes, Co-encadrant
M. Florian PEBAY-PEYROULA
Ingénieur de Recherche au CEA-Leti, Co-encadrant

Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle

Mots-clés : Sécurité matérielle, Injection de fautes, Modélisation de fautes, Architecture des processeurs, Analyse statique

---

Fault modeling using the RTL description of microarchitectures for joint hardware-software vulnerability analysis

Keywords: Hardware security, Fault injection, Fault Modeling, Processor architecture, Static analysis

Remerciements

Je souhaiterais en premier lieu remercier Vincent Beroulle, directeur de thèse, pour sa disponibilité et son investissement tout au long de cette thèse, tant sur le plan recherche que sur le plan enseignement ; et pour son encadrement rigoureux mais toujours dans la bonne humeur.

J'aimerais ensuite remercier Christophe Deleuze et Florian Pebay-Peyroula, co-encadrants de thèse, pour leur suivi et leurs conseils avisés durant nos (nombreuses) réunions. L'expertise de chacun a beaucoup enrichi le travail présenté dans ce manuscrit.

Je voudrais également remercier Lilian Bossuet et David Hély qui ont accepté de diriger le Comité de Suivi Individuel qui s'est réuni chaque année et m'a apporté un regard différent sur mon travail.

Merci à Karine Heydemann et Jean-Max Dutertre d'avoir accepté d'être rapporteurs pour cette thèse, ainsi qu'à Bruno Rouzeyre et Lilian Bossuet (une nouvelle fois) pour leur présence au jury.

Je voudrais aussi adresser mes remerciements à Thanos, pour ses conseils et son aide, particulièrement à mes débuts, pour prendre en main ce sujet complexe qu'est l'injection de faute.

Merci à Mahdi, Sam et Zaruhi dont les travaux pendant leur stage m'ont permis de maintenir le cap quand la motivation venait à manquer.

Merci à tous mes amis et collègues du LCIS, tout particulièrement de l'équipe CTSYS : Cyril, Baptiste et Elnaz qui ont commencé leur thèse en même temps que moi et qui arrivent également au bout du chemin ; la team Basket : Rina, Raphaël, Kostas et Yoann ; Rahul, Ehsan, Johan M., Caroline, Romain, Carole, Stéphanie, Laurent, Cédric, Amir, Arthur, Mickaël, Eskandar, Igyso, Karem, Afef, et tous les autres (la liste est longue) que j'ai pu côtoyé pendant cette thèse.

Pour finir, merci à toute ma famille qui m'a soutenu tout au long de cette expérience de trois ans et qui a toujours su me redonner le moral dans les moments difficiles.

Résumé

La sécurité numérique est aujourd'hui un enjeu majeur dans nos sociétés. Communications, énergie, transport, outils de production, Internet des Objets… Les systèmes numériques se multiplient et deviennent toujours plus critiques pour le bon fonctionnement du monde. Depuis un peu plus d'une vingtaine d'années, une nouvelle menace a émergé pour attaquer les systèmes : l'injection de faute. Elle consiste essentiellement à perturber un circuit pendant son fonctionnement, par diverses méthodes comme des perturbations sur l'alimentation du circuit, l'injection électromagnétique, ou l'injection laser ; afin de provoquer des erreurs. Ces erreurs peuvent ensuite être exploitées par un attaquant pour révéler des informations secrètes du circuit, ou passer outre des mesures de sécurité.

La complexification des systèmes numériques et les avancées technologiques comme la finesse de gravure rendent particulièrement vulnérables les systèmes numériques face aux attaques par injection de fautes. Pour contrer ces attaques efficacement et à un coût raisonnable, il est nécessaire de penser la sécurité dès la phase de conception du système. Pour cela, il faut comprendre précisément l'impact de ces fautes sur les processeurs. Les effets induits peuvent être modélisés à différents niveaux d'abstraction. Actuellement, si l'impact des fautes est relativement bien connu au niveau matériel, leurs effets au niveau logiciel restent mal compris. Les analyses de vulnérabilité au niveau logiciel se basent donc sur des modèles de faute logiciels simples que sont par exemple le saut d'instruction, la corruption de registre ou l'inversion de test. Ces modèles sont appliqués sans réelle prise en compte de la microarchitecture du processeur attaqué. Cette non-considération de l'aspect matériel pose la question du réalisme des modèles logiciels, qui conduit à deux types de problèmes : certains effets modélisés ne correspondent pas à des vulnérabilités réelles ; et, à l'inverse, certains effets affaiblissant la sécurité ne sont pas modélisés. Ces difficultés se transposent ensuite dans des contremesures sur-dimensionnées, ou, plus grave, sous-dimensionnées.

Pour lutter contre ces limitations des modèles de faute logiciels usuels, une étude précise de la microarchitecture des processeurs est requise. Dans cette thèse, nous explorons tout d'abord en quoi différentes structures du processeur, comme le pipeline ou les optimisations de forwarding et d'exécution spéculative, peuvent influer sur le comportement des fautes au sein du processeur et en quoi ces structures peuvent mettre à mal une vision purement logicielle de l'impact des fautes sur l'exécution d'un programme. Des injections au niveau RTL dans un processeur d'architecture RISC-V sont effectuées pour montrer que ces effets pourraient être exploités pour attaquer des contremesures logicielles typiques, ou encore une application de vérification de PIN sécurisée. Dans un deuxième temps est développée une méthode pour étudier plus généralement les effets des fautes dans un processeur. Cette méthode a un intérêt double. Le premier est la modélisation de fautes au niveau logiciel, avec notamment la définition de métriques d'évaluation des modèles. Le second est de conserver un lien avec le niveau RTL afin de pouvoir concrétiser les effets obtenus au niveau logiciel. Pour terminer cette thèse, nous étudions la possibilité d'utiliser des méthodes d'analyse statique pour analyser la sécurité de programmes face aux modèles de faute logiciels définis précédemment. Une analyse par interprétation abstraite et une analyse par exécution symbolique sont abordées.

Cette thèse, financée par l'IRT Nanoelec pour le projet Pulse, a été réalisée au sein du laboratoire LCIS de Valence, en collaboration avec le CEA-Leti de Grenoble. Elle a été dirigée par Vincent Beroulle (LCIS) et co-encadrée par Christophe Deleuze (LCIS) et Florian Pebay-Peyroula (CEA-Leti).

Abstract

Nowadays, digital security is of major importance to our societies. Communications, energy, transport, means of production, Internet of Things… The use of digital systems is ever increasing, making them critical to the correct working of our world. A little more than two decades ago, a new form of attack has risen: fault injection. Essentially, it consists in perturbing a circuit during computation, using various methods such as power glitches, electromagnetic injection or laser injection; in the aim of generating errors. These errors can then be exploited by an attacker to reveal secret information from the circuit, or to bypass some security measures.

System complexification and technological advances make digital systems particularly vulnerable against fault injection attacks. In order to thwart these attacks effectively and at a reasonable cost, it is necessary to consider security from the early phases of the design flow. To do that, a better understanding of how faults impact processors is required. Effects provoked by fault injection can be modeled at various levels of abstraction. Currently, if the impact of faults at the hardware level is relatively well known, the same cannot be said for the software level. Security analyses at the software level are based on simple software fault models such as instruction skip, register corruption or test inversion. These models are applied without any serious consideration for the microarchitecture of the attacked processor. This brings the question of the realism of these models, leading to two types of problems: some modeled effects do not correspond to actual attacks; and, conversely, some effects lowering the security of the system are not modeled. These issues then translate to over-engineered, or, worse, under-engineered countermeasures.

To face the limitations of typical software fault models, a precise study of processor microarchitectures is necessary. In this thesis, we first explore how various structures of the processor, such as the pipeline or optimization structures like forwarding and speculative execution, can influence the behavior of faults in the inner working of the processor; and how they call into question a pure software vision of how faults impact software execution. RTL injections are conducted in a RISC-V processor, to demonstrate how these effects could be exploited to counter typical software countermeasures and a hardened program that check PIN codes. Then, a method to study more generally the effects of faults in a processor is developed. The point of this method is twofold. The first is about modeling faults at the software level, with the definition of several metrics to evaluate models. The second point is about keeping a link to the RTL level, in order to be able to materialize effects obtained at the software level. Finally, to end this thesis, we study the possibility to use static analysis to analyze the security of programs against software fault models defined previously. Two methods are considered, one using abstract interpretation, and the other using symbolic execution.

This thesis, financed by the IRT Nanoelec for the Pulse project, has been conducted within the LCIS laboratory in Valence, in collaboration with the CEA-Leti in Grenoble. It has been supervised by Vincent Beroulle (LCIS), and co-supervised by Christophe Deleuze (LCIS) and Florian Pebay-Peyroula (CEA-Leti).

Table des matières

Chapitre I. Contexte et motivations

Résumé du chapitre

Ce premier chapitre introductif présente une vue d'ensemble du domaine de la sécurité numérique, et plus particulièrement de la sécurité matérielle et de ses enjeux. Nous verrons quels sont les grands types d'attaques matérielles, leur mise en œuvre et leurs buts respectifs. Puis nous nous concentrerons sur les problématiques soulevées par les attaques par injection de faute, avant de présenter les objectifs de cette thèse.

Sommaire

I.1. Introduction du chapitre

De nos jours, nombre d'aspects des sociétés modernes sont régis par des systèmes numériques. Energie, transport, communications, outils de production ; tous ces domaines intègrent une part grandissante de numérique et sont, en outre, de plus en plus interconnectés. Cette importance prise par les systèmes numériques en fait des cibles idéales pour des acteurs malveillants cherchant à s'enrichir, ou à générer des déstabilisations économiques ou politiques.

C'est pourquoi la sécurité des systèmes numériques est aujourd'hui un enjeu déterminant au bon fonctionnement de nos sociétés. Depuis quelques années, des attaques de grande ampleur émaillent régulièrement les gros titres des journaux, comme l'attaque Wannacry en mai 2017. Des mots tels que virus, malware, ransomware, cheval de Troie sont aujourd'hui bien connus du grand public. Toutes ces attaques entrent dans la catégorie des attaques logicielles : elles visent en général à exploiter des erreurs de conception dans les programmes ou les protocoles.

Mais les attaques peuvent également prendre d'autres formes, moins médiatisées. Un système numérique est composé d'une application logicielle qui est exécutée sur une architecture matérielle, le processeur. Or les processeurs peuvent être directement cibles d'attaques physiques, appelées attaques matérielles. A l'instar des attaques logicielles qui exploitent des faiblesses logicielles, les attaques matérielles mettent à profit les faiblesses du matériel. La sécurité matérielle a pour but de protéger face à ces attaques. Elle vise par exemple à la sécurisation d'algorithmes de cryptographie, la protection face à la rétro-ingénierie, ou la sécurisation des secrets lors des processus de fabrication des puces.

Les attaques matérielles nécessitent en général un accès physique au circuit attaqué, ce qui peut en limiter la portée. Pour autant, cela serait une erreur de négliger cet aspect car le niveau global de sécurité des systèmes d'information est défini par le niveau de sécurité du maillon le plus faible. Si une partie d'un système est peu sécurisée, c'est probablement de cette partie que viendront les attaques. Les moyens d'action permis par les attaques matérielles sont moins spectaculaires, mais n'en sont pas moins efficaces.

D'autre part, l'essor de l'Internet des Objets, avec le déploiement de milliards d'objets connectés, et plus généralement le contexte des systèmes embarqués rend plus plausibles les attaques matérielles en multipliant l'accessibilité des cibles potentielles. L'interconnexion de tous ces objets fait que l'attaque de quelques nœuds peut parfois mettre en danger l'intégralité du système.

Ainsi il est nécessaire de protéger les systèmes numériques, et en particulier les processeurs, face au risque présenté par les attaques matérielles. Dans le reste de ce chapitre, nous présenterons plus précisément ce que sont ces attaques matérielles, puis nous évoquerons la phase de protection des systèmes, avant de définir les objectifs de la thèse.

I.2. Attaques matérielles

I.2.1. Attaques par canaux cachés

Le premier grand type d'attaque matérielle est l'attaque par canaux cachés, également appelée attaque par canaux auxiliaires. Ce sont des attaques passives qui consistent à observer certaines caractéristiques d'un circuit pendant son fonctionnement, pour en déduire des informations secrètes. Typiquement, le but recherché par un attaquant est de trouver la clé utilisée pour chiffrer un message.

Plusieurs grandeurs peuvent être observées, mais les plus communes sont la consommation électrique du circuit, ses émissions électromagnétiques, ou encore la durée d'un calcul [26]. Dans des implémentations naïves d'algorithmes de cryptographie, ces grandeurs peuvent être directement corrélées à la valeur de la clé de chiffrement. Il suffit alors d'observer par exemple les pics de consommation/émission dans le temps pour directement déduire la valeur des bits de la clé, comme le montre la Figure 1.

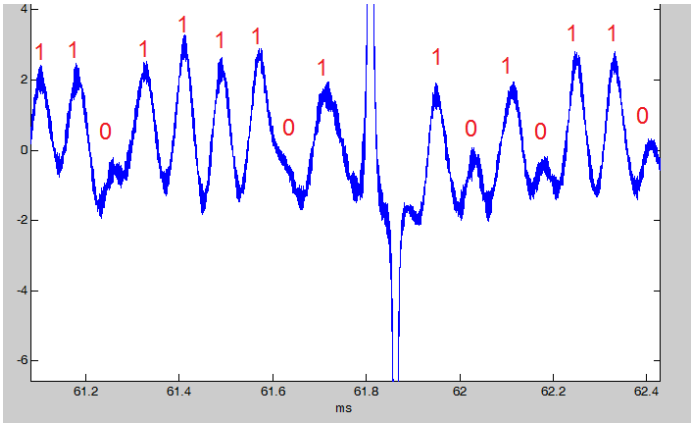

FIGURE 1 : EXEMPLE DE SIGNAL DIRECTEMENT CORRELE AVEC LA VALEUR DE LA CLE DE CHIFFREMENT

Plusieurs mesures de sécurité peuvent être prises pour réduire la quantité d'information fuitant dans les canaux auxiliaires, mais il existe également des méthodes d'attaque plus sophistiquées, comme la DPA (Differential Power Analysis) [37] ou la CPA (Correlation Power Analysis) [12], qui ne seront pas développées ici.

I.2.2. Attaques par injection de faute

Le second grand type d'attaque matérielle, qui fait l'objet particulier de cette thèse, est appelé attaque par injection de faute, ou encore attaque par perturbation. Contrairement à la première forme d'attaque matérielle, il s'agit là d'attaques actives : le but de l'attaquant est de perturber le circuit pendant son fonctionnement, comme représenté sur la Figure 2. Ces perturbations peuvent modifier l'état du système et l'amener à un état exploitable par un attaquant. Les modifications induites dans le processeur sont appelées fautes car elles engendrent des états anormaux du processeur. Les attaques par injection de faute sont en général non destructives ; leur but n'étant pas de dégrader le circuit de manière permanente, mais de simplement le perturber momentanément.

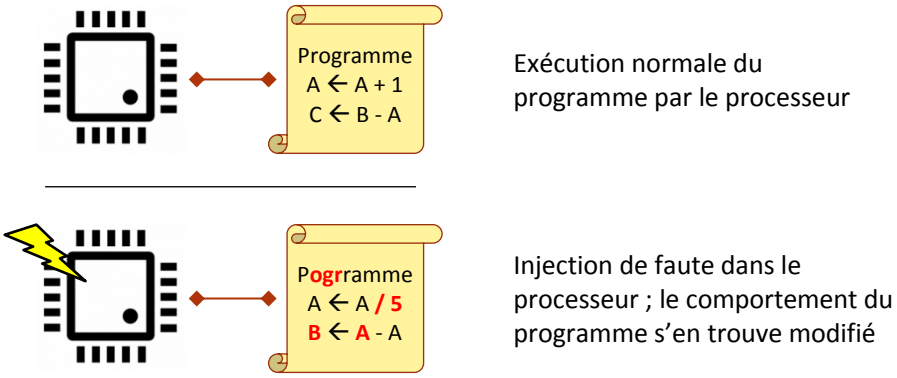

FIGURE 2 : PRINCIPE DE L'INJECTION DE FAUTE

Le principe de cette attaque est apparu en 1997 dans les travaux de Boneh et al. [10], qui proposent d'attaquer des algorithmes de cryptographie comme RSA par cette méthode. La problématique des fautes était déjà connue auparavant dans le domaine de la sûreté, notamment le domaine aérospatial où les systèmes, moins protégés par notre atmosphère, peuvent être perturbés par des chocs de particules venant par exemple du Soleil. Boneh et al. proposent pour la première fois d'appliquer le même principe dans le domaine de la sécurité, c'est-à-dire de générer sciemment des fautes pour casser la sécurité de systèmes numériques.

I.2.2.1. Moyens d'injection

Il existe différentes méthodes pour perturber un circuit. Une première méthode consiste à faire sortir un circuit de son régime de fonctionnement normal. Parmi les moyens les plus connus, nous trouvons des perturbations de la tension d'alimentation ou de la fréquence d'horloge. Le premier cas consiste par exemple à sous-alimenter momentanément le circuit [4], ou lui envoyer une impulsion de tension négative [69]; le second à augmenter transitoirement sa fréquence d'utilisation [59]. Le circuit n'étant pas fait pour fonctionner sous ces conditions, son comportement peut s'en trouver modifié. Dans le même ordre d'idée, il est possible aussi de chauffer le circuit pour modifier ses caractéristiques [38]. Toutes ces méthodes permettent d'impacter le circuit de manière assez

globale, créant aléatoirement, en première approximation, des fautes dans l'ensemble du circuit (des méthodes d'analyse plus sophistiquées permettent cependant de montrer que certaines structures sont plus sensibles que d'autres [67]).

Une deuxième manière d'injecter des fautes est d'impacter le circuit avec des ondes électromagnétiques. On parle d'injection électromagnétique lorsqu'une antenne est approchée du circuit pour envoyer une impulsion électromagnétique [47]. Cette méthode d'injection permet des attaques plus localisées que les moyens évoqués précédemment car la position de l'antenne détermine les structures du processeur qui seront plus probablement touchées. Un moyen d'attaque similaire est l'injection au laser [60,66]. Dans ce cas, une partie précise du circuit est illuminée pendant un bref instant, générant des effets dans le silicium de la puce, sur lesquels nous reviendrons. Le laser permet des injections encore plus ciblées que les injections électromagnétiques, mais suppose des moyens d'attaques plus sophistiqués, puisqu'il est au préalable nécessaire de décapsuler la puce pour avoir un accès direct au silicium. Cette méthode est donc assez puissante, mais est très intrusive et suppose une certaine expertise de l'attaquant.

I.2.2.2. Différents types d'attaques

Par injection de faute, un attaquant peut rechercher différents buts. Un des objectifs peut être d'extraire de l'information secrète du circuit. Comme pour les attaques par canaux auxiliaires, les algorithmes de cryptographie sont souvent visés. Pour révéler de l'information secrète, plusieurs méthodes sont possibles. Parmi les plus utilisées, l'*analyse de faute différentielle* (DFA) consiste à comparer les résultats d'une exécution fautée avec les résultats d'une exécution saine d'un même algorithme [9]. Si les effets de l'injection sont suffisamment maîtrisés, les différences observées peuvent permettre de remonter à la clé secrète du système. Un exemple d'une telle attaque est présenté dans [46]. Un second objectif peut être d'extraire de l'information secrète par une attaque en *safe-error*. Dans ce cas, ce ne sont pas les effets précis de la faute sur le système qui sont observés, mais simplement le fait que la faute ait eu un effet ou non [62]. Si la faute ne fait pas dévier l'exécution, l'attaquant en déduit quelque chose ; et de même, si la faute fait dévier l'exécution, l'attaquant en déduit également quelque chose, indépendamment de l'impact précis obtenu. Enfin, un troisième objectif possible d'un attaquant peut être de passer à travers des mesures de sécurité, par exemple en modifiant le flot de contrôle de l'exécution d'un programme.

Par ailleurs, les attaques peuvent être simples (une seule attaque) ou multiples. Avec un moyen d'injection localisé (laser ou injection électromagnétique), il est possible d'attaquer simultanément plusieurs endroits du circuit pour toucher plusieurs structures spécifiques en même temps. Les attaques peuvent aussi être successives, pour, par exemple, impacter le circuit tout en déjouant des mécanismes de sécurité. Globalement, les attaques multiples ajoutent un degré de complexité à l'analyse de sécurité.

I.3. Contremesures et modélisation des injections de fautes

Les attaques par injection de fautes sont donc des menaces sérieuses ; et elles le sont d'autant plus avec l'évolution des technologies. En approchant les technologies de gravure toujours plus près des limites permises par la physique, les puces deviennent en même temps plus sensibles aux perturbations extérieures. Parallèlement à cela, les moyens disponibles pour les attaquants évoluent et deviennent toujours plus puissants et accessibles.

Pour lutter contre la menace représentée par les injections de fautes, il est donc nécessaire de protéger les systèmes avec des contremesures, qu'elles soient au niveau matériel (introduction de structures de détection ou correction dans la puce) ou au niveau logiciel (moyens de protection dans le programme) ; ou même dans les deux. Ces mesures de sécurité ont évidemment un coût, qui peut être assez considérable en fonction des options choisies. Ce coût provient tout d'abord de la phase de conception des contremesures. Il est relativement facile d'intégrer et de tester diverses contremesures logicielles, mais cela n'est pas forcément le cas pour les contremesures matérielles. La durée et le coût de fabrication des puces rend les analyses de sécurité avec des injections réelles assez coûteuses ; une détection tardive des vulnérabilités entrainant des surcoûts importants. Pour réduire ces coûts, il est nécessaire de penser la sécurité plus tôt dans le flot de développement du système, en ayant recours à une modélisation des effets induits par les injections de fautes.

La difficulté réside alors dans cette phase de modélisation. La complexification des processeurs rend cette étape compliquée car les effets observés sur le programme à exécuter sont très variés. Or, une mauvaise compréhension des effets des fautes conduit à sous-dimensionner ou sur-dimensionner les contremesures, ce qui impacte soit la sécurité du système, soit son coût et ses performances.

I.4. Objectifs de la thèse

Pour aboutir à une sécurité satisfaisante face aux attaques par injection de faute, il est nécessaire d'avoir au préalable une modélisation des fautes précise, qui prenne en compte le système dans son ensemble, c'est-à-dire qui considère conjointement l'architecture matérielle et l'application logicielle. Nous considérerons pour cela que la description RTL du processeur est disponible, et nous étudierons son comportement en simulation. Nous reviendrons sur les raisons de ces choix dans le chapitre suivant.

La meilleure compréhension des effets des fautes doit permettre de développer des analyses de sécurité plus pertinentes, qui sont primordiales pour pouvoir anticiper les menaces et prévoir des contremesures adaptées. Pour mener à bien ces analyses de sécurité, des techniques d'analyse statique de programme (c'est-à-dire analyse sans exécution du programme) sont de plus en plus utilisées. Nous nous tournerons donc vers ces techniques qui permettent, au prix d'approximations contrôlées, de raisonner sur des ensembles d'exécutions avec des paramètres et des fautes variables.

Pour résumer :

- Proposer une approche qui permette une meilleure compréhension de l'impact de fautes sur un système composé d'une partie matérielle, le processeur, et d'une partie logicielle, le programme à exécuter.
- Utiliser cette modélisation précise pour permettre des analyses statiques de sécurité qui puissent identifier les vulnérabilités du système afin de les corriger au plus juste coût.

Ces deux points représentent les objectifs généraux de la thèse. Pour les préciser, une étude de l'état de l'art est nécessaire. C'est l'objet du chapitre suivant. A l'issue de cette revue seront exposés la problématique précise traitée dans cette thèse, ainsi que le plan adopté pour y répondre.

Chapitre II. Etat de l'art et problématique

Résumé du chapitre

Dans ce chapitre, nous proposons une vue d'ensemble de l'état de l'art de l'injection de faute, tant au niveau matériel qu'au niveau logiciel. Plusieurs aspects de l'injection sont développés, notamment la modélisation des fautes, l'utilisation de ces modèles pour mener des études de sécurité et les contremesures typiquement utilisées pour lutter contre ce type d'attaque.

Après cette revue de l'état de l'art, nous exposons en quoi l'analyse de l'architecture des processeurs pourrait améliorer la modélisation logicielle des effets induits par les fautes au niveau matériel, aboutissant ainsi à des analyses de sécurité plus pertinentes. Nous proposons, pour finir, le plan adopté dans cette thèse pour répondre à cette problématique.

Sommaire

II.1. Introduction du chapitre

Comme nous l'avons évoqué dans le chapitre précédent, les attaques par injection de faute constituent un vecteur d'attaque possible pour les systèmes numériques. La recherche en sécurité matérielle a déjà abouti à un certain nombre de résultats pour analyser l'impact de ces fautes sur un système et le sécuriser ; résultats qu'il est nécessaire de passer en revue afin de définir précisément l'angle de cette thèse.

Dans ce chapitre, nous nous intéresserons donc à la manière d'étudier la sécurité des circuits face aux attaques par injection de faute. Nous commencerons par appréhender l'aspect matériel de la chose, avec une discussion sur la modélisation matérielle des fautes, puis sur la façon de réaliser des injections avec ces modèles, pour finir par la présentation de quelques types de contremesures matérielles. Dans un second temps, nous nous attarderons sur l'aspect logiciel, avec une trame similaire : modélisation logicielle, puis analyse de programmes avec des modèles de faute logiciels, et enfin quelques exemples de contremesures logicielles. Après ce tour d'horizon des méthodes utilisées pour l'étude de l'injection de fautes, nous définirons la problématique traitée dans cette thèse et proposerons un plan pour y répondre.

II.2. Injection de faute matérielle

L'injection de faute étant une attaque impactant directement le côté matériel du système, il est tout naturel de commencer par décrire sa modélisation au niveau matériel.

Avant cela, il est utile de noter que cette thèse traite uniquement de fautes dites transitoires, c'est-à-dire de fautes dont les effets disparaissent complètement si le système est remis à zéro (ces fautes sont à opposer aux fautes dites permanentes). En effet, dans le domaine de la sécurité, le but d'un attaquant est généralement de perturber momentanément un système, sans l'endommager. La dégradation permanente d'un système peut être un objectif d'un attaquant, par exemple pour invalider indéfiniment une protection, mais ce type d'attaque peut être risqué car le circuit peut aussi être rendu inutilisable. Monter une telle attaque peut se révéler coûteux et est donc moins courant.

II.2.1. Modélisation matérielle des fautes

Au niveau matériel, les fautes peuvent être modélisées à différents niveaux d'abstraction et la modélisation diffère en fonction du moyen d'injection utilisé [23]. Prenons le cas d'injections laser. Une illumination laser sur le silicium provoque la création de paires électron-trous [31]. Ces paires, en se recombinant, créent des courants électriques transitoires qui influent sur l'état de conduction ou de blocage de transistors du circuit [18]. Ceci peut impacter la logique séquentielle, au même titre que la logique combinatoire. Dans cette thèse, nous nous intéressons aux impacts sur la logique séquentielle, sur des éléments mémoire appelés bascules car les effets sur le programme doivent à un moment être capturés dans au moins une de ces bascules. Les modifications induites dans les bascules du circuit prennent trois formes différentes qui sont le *bit-set* (la valeur de la bascule est forcée à 1), le *bit-reset* (la valeur de la bascule est forcée à 0), ou le *bit-flip* (la valeur de la bascule est inversée). Au niveau RTL, ces trois modèles sont très utilisés.

La modélisation des fautes au niveau RTL induit forcément une perte de précision par rapport aux effets réels d'une attaque. Ceci est dû à plusieurs facteurs, comme la difficulté à modéliser la méthode d'injection utilisée (perturbation de l'alimentation, laser, etc) ; mais tient aussi au fait que le niveau RTL abstrait certains détails d'implémentation du circuit. Toutefois, la modélisation RTL

reste très utilisée car elle offre une précision relativement bonne et permet de comprendre simplement les effets des fautes, et ce, tôt dans le flot de développement [41]. Dans cette thèse, nous considérons donc le niveau RTL comme le niveau de référence ; nous posons l'hypothèse que l'injection de faute crée des bit-flips/bit-sets/bit-resets, sans nous préoccuper des effets aux niveaux d'abstraction plus bas.

Une injection de faute peut impacter une seule, ou plusieurs bascules. Nous parlerons respectivement de faute simple-bit et faute multiple. La multiplicité des fautes dépend notamment de la technologie du circuit, les nouvelles technologies en microélectronique rendant les circuits plus sensibles aux perturbations extérieures ; et du moyen d'injection : les méthodes d'injection précises spatialement comme l'injection laser ou, à moindre mesure, l'injection électromagnétique auront plus tendance à créer des fautes simple-bit que les méthodes d'injection plus globales comme les perturbations en tension ou les perturbations d'horloge. Les fautes multiples sont complexes à étudier à cause de l'explosion combinatoire qu'elles induisent. Certaines méthodes visent à gérer cette complexité en considérant que les différentes bascules ne sont pas touchées totalement aléatoirement, mais en fonction de la fonctionnalité du circuit [52]. Dans le même ordre d'idée, certains modèles considèrent que tous les bits d'un même mot peuvent être touchés simultanément ; il a en effet été montré que ce type d'attaque pouvait être mené expérimentalement [24]. Notons que malgré les avancées dans les techniques de fabrication des puces, les injections simple-bit restent une menace sérieuse car il a été montré qu'elles étaient encore possibles sur des technologies avancées comme la technologie CMOS 28nm, par injection laser, et qu'elles pouvaient même représenter 50% des fautes injectées [22].

II.2.2. Injection de fautes au niveau RTL

Une fois le modèle de faute défini, il est nécessaire, pour analyser la sécurité d'un système, de choisir une méthode d'injection. Plusieurs méthodes sont possibles, à différents niveaux d'abstraction [68].

Par rapport aux injections avec des moyens réels, les injections RTL permettent, au prix de quelques approximations, un meilleur contrôle et une meilleure observabilité : les fautes peuvent être précisément injectées dans les structures à étudier et la propagation de ces fautes peut être observée tout aussi précisément. Ces caractéristiques amènent également la reproductibilité des fautes, là où des injections réelles ont une part de variabilité. Tous ces points font du niveau RTL un niveau d'abstraction adéquat pour comprendre l'impact des fautes sur un système.

II.2.2.1. Simulation RTL

Au niveau RTL, une des méthodes les plus utilisées est la simulation. En simulation, l'injection de fautes peut elle-même se faire de plusieurs manières [28], à savoir par l'utilisation de saboteurs, de mutants, ou de commandes de l'outil de simulation.

Un saboteur est un composant au niveau RTL, ajouté au circuit original et dont le rôle est de modifier la valeur d'un ou plusieurs signaux pendant le fonctionnement de la simulation. L'activation de ce composant est contrôlée par l'utilisateur, par l'envoi d'un signal au saboteur.

Un mutant est un composant au niveau RTL, qui remplace un composant original et qui peut, si l'utilisateur lui envoie un signal d'activation, se comporter comme si une faute avait été injectée. La différence avec un saboteur est que la modification se situe à l'intérieur du circuit alors que le saboteur se situe à l'extérieur.

Enfin, la dernière solution est d'utiliser les commandes de l'outil de simulation RTL. Ici, pas besoin de modifier le circuit à simuler. Un simulateur possède des commandes qui permettent de forcer les signaux à une valeur définie par l'utilisateur. L'injection de faute RTL avec cette solution est bien plus simple à mettre en œuvre car elle ne nécessite aucune modification du circuit à simuler. Les modèles de faute matériels pouvant être utilisés sont cependant plus restreints car les commandes d'un simulateur ne permettent pas autant d'expressivité qu'un composant décrit dans un langage de description matériel tel que le VHDL (mais ceci dépend du simulateur utilisé).

Outre la simulation RTL directe, il existe aussi des outils d'injection qui permettent d'automatiser les campagnes d'injection, tels que l'outil MEFISTO [33] ou l'outil VERIFY [61].

II.2.2.2. Emulation RTL

Le principal inconvénient des simulations au niveau RTL est leur relative lenteur, par rapport à des injections réalisées directement au niveau logiciel. Il existe une alternative pour injecter des fautes au niveau RTL, qui permet d'accélérer les campagnes d'injection : l'émulation [39]. Cette méthode consiste à instrumenter le circuit à analyser avec des saboteurs ou des mutants, puis l'implémenter sur un (ou plusieurs) FPGA. Les injections peuvent ensuite être réalisées directement au niveau matériel en faisant fonctionner le FPGA, plutôt qu'avec un logiciel de simulation sur un ordinateur hôte. L'émulation est adaptée pour des campagnes d'injection assez conséquentes. Son défaut est d'être plus complexe à mettre en œuvre qu'une simple simulation et surtout de perdre en flexibilité. En simulation, l'expérimentateur a un contrôle total sur le système, il peut à tout moment injecter ou observer où bon lui semble alors que l'émulation est plus rigide (une modification des paramètres d'injection nécessite une nouvelle synthèse, placement-routage, etc). C'est avant tout pour cette flexibilité d'utilisation que la simulation a été retenue dans cette thèse.

II.2.3. Contremesures matérielles

Après avoir injecté des fautes et potentiellement trouvé des vulnérabilités, vient la phase de protection du système. Des contremesures peuvent être ajoutées au niveau matériel ou au niveau logiciel. Nous étudierons ici quelques contremesures matérielles ; l'étude des contremesures logicielles étant reléguée dans une section ultérieure. Cette vue d'ensemble des contremesures matérielles ne se veut pas exhaustive mais permet de donner une idée des types de protection utilisés et de leur coût.

Les contremesures matérielles sont généralement basées sur un principe de redondance [44]. Il faut distinguer en particulier la redondance matérielle, la redondance temporelle et la redondance de l'information.

II.2.3.1. Redondance temporelle

La redondance temporelle consiste essentiellement à effectuer plusieurs fois de suite les mêmes opérations sur le processeur, pour ensuite comparer leur résultat. Si ces résultats sont différents, c'est qu'au moins une des exécutions a été attaquée. Ce type de contremesure peut être envisagé au niveau matériel, mais il est plutôt utilisé du côté logiciel, comme nous le verrons par la suite. Cette contremesure est efficace dans le sens où elle peut détecter beaucoup d'attaques ayant un impact sur le résultat final (la valeur comparée), mais elle engendre un surcoût important puisque le temps d'exécution est multiplié par le nombre d'exécutions effectuées.

II.2.3.2. Redondance matérielle

La redondance matérielle est similaire à la redondance temporelle, à ceci près que les différentes exécutions ne sont pas réalisées en série sur un même circuit, mais sur des blocs matériels distincts exécutant les mêmes opérations en parallèle. Le choix se porte en général sur la duplication ou la triplication. Dans le premier cas, le même circuit est instancié deux fois et les résultats sont comparés pour simplement détecter des erreurs. Dans le second cas, le circuit est instancié trois fois et la comparaison des résultats est en général remplacée par un système de vote : chacune des trois exécutions propose un résultat et le résultat majoritaire est retenu pour la suite de l'exécution. Ce système permet donc de corriger une faute, au lieu de simplement la détecter ; ou de détecter deux fautes différentes.

Les avantages et inconvénients de la redondance matérielle sont similaires à ceux de la redondance temporelle, mais le coût de la contremesure est déplacé : plutôt que multiplier la durée d'exécution, c'est la surface de silicium nécessaire qui est multipliée. Par ailleurs, le bloc matériel responsable de la comparaison des résultats ou du système de vote peut également avoir un impact négatif sur la fréquence de fonctionnement du circuit en allongeant les chemins critiques du circuit.

Les contremesures basées sur la redondance temporelle ou matérielle, si elles ont un intérêt évident d'un point de vue théorique, ne sont néanmoins pas parfaites dans des conditions réelles. Injecter les mêmes fautes dans deux exécutions consécutives d'un même calcul, ou dans deux blocs matériels identiques est loin d'être une chose improbable, surtout avec des moyens d'injection avancés tels que l'injection électromagnétique ou l'injection laser. Il est donc envisageable qu'un attaquant puisse fausser de la même manière les différentes exécutions et ainsi passer outre ces contremesures. D'autres variantes de duplication existent [3], pour rendre cette double attaque plus compliquée. Nous pouvons par exemple citer la duplication complémentaire, duplication matérielle où le second bloc opère sur des données complémentées. Ceci peut être intéressant suivant le moyen d'injection/le modèle de faute matériel considéré (bit-set ou bit-reset par exemple). Remarquons également que dans le cas où l'attaquant est capable de générer deux fautes identiques dans deux exécutions différentes, la triplication avec un système de vote n'est pas plus performante que la duplication, car les résultats issus des fautes sont majoritaires. Le surcoût engendré par la triplication par rapport à la duplication, est dans ce cas non justifié. D'autre part, toujours dans le cadre d'attaques doubles, la triplication avec correction d'erreur se révèle plus problématique qu'une triplication avec simple détection d'erreur. L'introduction d'un système de correction d'erreurs peut donc être, dans le cadre de la sécurité, plus préjudiciable qu'une simple détection d'erreur.

Enfin, il est utile de noter que pour diminuer le coût relativement élevé des contremesures basées sur la réplication, il est possible d'appliquer ces solutions sélectivement à certains endroits critiques du circuit, au prix bien sûr d'une portée plus faible de la contremesure.

II.2.3.3. Redondance de l'information

La redondance de l'information peut prendre plusieurs formes. Nous distinguons ici les méthodes basées sur des codes correcteurs d'erreur et celles basées sur un composant matériel couramment appelé *watchdog*.

Codes correcteurs d'erreur

La première forme de redondance de l'information consiste à adjoindre à chaque donnée manipulée par le processeur, un ou plusieurs bits de contrôle. Nous pouvons considérer ici des solutions comme le bit de parité ou les codes correcteurs d'erreur.

Le bit de parité consiste simplement à adjoindre un bit à chaque donnée de manière à ce que le nombre de bits à l'état haut dans le mot soit pair. Au moment de l'utilisation de cette donnée, la parité est vérifiée ; si elle n'est pas respectée, c'est qu'une faute a été injectée. Cette solution permet de détecter une faute simple-bit dans un mot de donnée.

Les codes correcteurs d'erreur sont des procédés plus avancés qui permettent non seulement de détecter des erreurs, mais également d'en corriger. Le code correcteur d'erreur le plus connu est le code de Hamming, mais il existe d'autres comme le code de Golay ou même des codes non-linéaires ou des codes cycliques.

La redondance de l'information est souvent utilisée pour protéger la mémoire ou le banc de registre, mais il est possible de les utiliser également dans le pipeline du processeur ; par exemple dans [25], Elliot et al. protègent les opérations d'une Unité Arithmétique et Logique (ALU) avec un code de Hamming. De même, Medwed et al. présentent une solution dans [45] à base de codes multi-residus. Ces derniers rapportent un surcoût global environ 25% plus faible, en surface, par rapport à une contremesure de duplication matérielle.

Contrôle par watchdog

Un *watchdog* (« chien de garde » en français) est un module matériel séparé du processeur, qui contrôle le bon fonctionnement de ce dernier. Ce système reçoit typiquement des informations préalables sur les applications à protéger, qu'il doit ensuite vérifier au moment de l'exécution de ces applications. Les *watchdogs* ont l'avantage d'être moins intrusifs que les solutions précédentes, dans la mesure où ils n'interfèrent pas avec la conception originale du processeur, mais agissent en parallèle du processeur. L'inconvénient principal est qu'il nécessite une phase de préparation, là où les codes correcteurs d'erreur ou la duplication sont transparents une fois implémentés.

Parra et al. présentent dans [53] un *watchdog* dont le rôle est de s'assurer de l'intégrité du flot de contrôle en vérifiant le flot d'instruction en entrée et sortie du pipeline du processeur. Un autre exemple se trouve dans [7], où Benso et al. proposent un *watchdog* pour vérifier à la fois la validité des lectures et écritures dans la mémoire et celle du flot de contrôle du programme.

II.2.3.4. Discussion sur les contremesures matérielles

Les contremesures présentées dans cette section ne sont pas exclusives ; plusieurs types de protection peuvent être combinés à divers endroits du système pour atteindre une sécurité plus optimale. Dans [42], Lindoso et al. utilisent par exemple un code correcteur d'erreurs pour protéger la mémoire et un *watchdog* pour vérifier le flot d'instruction du processeur. Dans sa thèse de Master [32], Heida propose de protéger un processeur RISC-V en dupliquant le pipeline mais en utilisant un code correcteur d'erreur pour la mémoire.

Il est à noter qu'en plus de ces mesures de protection qui agissent sur l'architecture du système, il existe aussi des contremesures qui permettent de détecter directement certains moyens d'injection. Citons notamment les détecteurs de lumière qui permettent de détecter une attaque laser par exemple. Ou encore les détecteurs de fréquence et de tension d'alimentation, qui détectent des

variations anormales de ces grandeurs. Enfin, il est possible de couvrir le circuit d'une couche de métal afin d'empêcher les attaques électromagnétiques (à moins bien sûr d'enlever cette couche de métal, ce qui n'est pas une opération triviale pour un attaquant).

De manière générale, les contremesures matérielles sont intéressantes, mais engendrent un surcoût important, particulièrement en surface de silicium. Ce surcoût peut être critique suivant les situations. Dans le domaine de l'Internet des Objets par exemple, des solutions à plus bas coût sont souvent préférables. L'une des caractéristiques des contremesures matérielles est que leur coût est indépendant des programmes exécutés par le processeur. Si une solution est adoptée, peu importe que les applications exécutées soient sensibles ou non, le coût de la protection reste le même. Si le processeur ne doit exécuter qu'une faible proportion d'opérations sensibles, alors ce coût peut être injustifié. Des solutions plus ciblées pourraient potentiellement permettre d'avoir un même niveau de sécurité, à un meilleur coût. Parmi ces solutions se trouvent les contremesures logicielles.

II.3. Injection de faute logicielle

Les protections logicielles sont par nature plus ciblées car elles n'agissent qu'à certains moments de l'exécution d'une application. Seulement, ce type de contremesure agit plus en aval : il vise à contrecarrer les effets des fautes visibles au niveau du programme (contrairement aux contremesures matérielles qui peuvent agir même sur des détails non visibles du point de vue logiciel). Pour pouvoir concevoir des protections efficaces, il est par conséquent nécessaire de comprendre l'impact des fautes à ce niveau d'abstraction.

Dans cette partie, nous étudierons donc l'injection de faute du point de vue logiciel. La structure sera la même que dans la partie précédente ; nous parlerons tout d'abord de modélisation et de méthodes d'injection, puis nous finirons sur la présentation de quelques contremesures logicielles.

II.3.1. Modélisation logicielle des fautes

Au niveau logiciel, la modélisation peut s'effectuer à différents niveaux, mais elle est considérée en général à deux niveaux : au niveau du code source ; ou au niveau objet, c'est-à-dire après compilation du code source. L'analyse au niveau objet est plus complexe qu'au niveau du code source car les transformations et optimisations effectuées par le compilateur peuvent obscurcir la compréhension du programme ; mais elle est aussi plus proche de la réalité car cela correspond exactement aux opérations effectuées par le processeur. Le compilateur modifiant la structure du programme, les conclusions tirées au niveau du code source peuvent ne plus être pertinentes une fois le code compilé. Dans cette thèse, nous ferons indifféremment mention de niveau objet ou niveau assembleur ; dans ce second cas, nous considérons le code désassemblé à partir du fichier objet.

La littérature scientifique regorge de modèles de faute logiciels différents ; certains étant des variantes d'autres. Mais globalement, ces modèles tombent dans une des catégories suivantes, répertoriées par la *Joint Interpretation Library* dans son interprétation des Critères Communs [34] :

- Modification d'une valeur lue en mémoire (également appelée corruption de registre)
- Modification de la valeur contenue à une adresse mémoire (également appelée corruption de mémoire)
- Remplacement d'une instruction par une autre
- Saut d'une instruction (ceci peut être dû à un remplacement d'instruction particulier par exemple [48])

- Inversion de test conditionnel (ceci peut être dû à plusieurs facteurs, comme un saut d'instruction ou la modification des valeurs comparées [57])
- Saut à un endroit du programme
- Erreur de calcul

Ces modèles sont globalement assez génériques et certains doivent être précisés pour pouvoir être utilisés. C'est le cas des modèles consistant à modifier certaines valeurs par exemple : quelles valeurs sont considérées ? Est-ce que la faute remplace une valeur par une valeur aléatoire ? Par une valeur spécifique ? Au vu de différents travaux, il semble que les valeurs utilisées soient en général 0 (tous les bits à 0) ou -1 (tous les bits à 1). Ces différentes possibilités peuvent elles-mêmes être appliquées soit à la donnée entière, soit à un octet seulement. D'autres valeurs sont moins courantes car souvent considérées moins réalistes (par exemple, un modèle qui remplacerait par la valeur 0x55 serait typiquement considéré improbable). Un autre exemple de modèle très générique est le remplacement d'instruction. Quelles sont les remplacements autorisés par le modèle ? Ici, pas de réponse claire ; ce modèle reste peu utilisé relativement aux autres modèles, car trop difficile à préciser.

L'intérêt de modéliser les fautes au niveau logiciel est de pourvoir s'extraire de considérations matérielles et de n'étudier que l'impact final de la faute. Ce plus haut niveau d'abstraction permet d'étudier plus facilement la robustesse des programmes. Mais cette caractéristique est à double tranchant : en ignorant le côté matériel du système, les modèles de fautes ne peuvent représenter qu'imparfaitement les effets réels des fautes matérielles. La volonté de représenter les effets des fautes par un ensemble restreint de modèles logiciels simples peut conduire à sous-estimer d'autres effets problématiques. Ce point est discuté plus en détail par la suite.

II.3.2. Analyse de programme au niveau logiciel

Une fois un (ou des) modèle(s) de faute logiciel(s) défini(s), l'analyse de sécurité d'une application peut débuter. Au niveau logiciel, de nombreuses méthodes d'analyse ont été développées.

Dans [11], Bréjon et al. proposent une méthode alliant analyse dynamique (avec exécution du programme) et statique (sans exécution du programme), avec notamment l'utilisation d'exécution symbolique, pour étudier la sécurité de programmes au niveau binaire. Deux types de modèles de faute logiciels sont utilisés : le saut d'instruction et la corruption de registre aléatoire. Trois métriques sont proposées pour évaluer la sensibilité du programme face à ces modèles de faute.

L'outil Lazart est décrit dans [57] pour étudier la robustesse de programmes face à des attaques visant le flot de contrôle, et en particulier face au modèle d'inversion de test conditionnel. Cette méthode est basée sur une décomposition du flot de contrôle en un graphe qui permet d'identifier les blocs qu'un attaquant aurait intérêt à attaquer. L'un des intérêts de cette méthode est de pouvoir prendre en compte des injections de fautes à différents moments d'une même exécution du programme.

Goubet et al. proposent dans [30] un système permettant de valider la pertinence de contremesures logicielles en comparant un programme protégé avec sa version non protégée. Les programmes sont d'abord représentés par des automates finis. Ces automates sont ensuite déroulés et chaque chemin d'exécution permet de construire une formule qui peut être résolue par un solveur de contraintes SMT (Satisfiability Modulo Theory) pour trouver les chemins qui mènent à des attaques réussies. Deux modèles de faute sont pris en compte, le saut d'instruction et le remplacement d'instruction.

Pour analyser la robustesse d'un programme, les méthodes d'analyses sont donc assez diverses. Plusieurs travaux se servent de méthodes d'analyse formelle pour arriver à des preuves mathématiques de sécurité. Notons toutefois que chaque méthode repose en grande partie sur la pertinence de quelques modèles de faute logiciels particuliers. Or, comme mentionné dans la section précédente, ces modèles de faute ne sont pas parfaits. Ils ne peuvent représenter qu'un sous-ensemble des impacts possibles des fautes sur le programme. Si les outils évoqués permettent d'aboutir à des preuves de sécurité efficaces, ces preuves sont conditionnées par la validité et la représentativité des modèles de faute logiciels considérés.

Pattabiraman et al. [54] proposent quant à eux un système d'analyse plus générique, au niveau assembleur. L'intérêt de cette méthode réside dans l'utilisation de modèles de faute représentés de manière symbolique : chaque erreur introduite dans le programme est représentée par un unique symbole spécial (par exemple, pour une corruption de registre le symbole représente toutes les valeurs possibles), propagé dans toutes les variables impactées. Cette méthode permet donc d'englober de nombreux modèles de faute logiciels différents. Dans cette méthode cependant, seul le symbole d'erreur est propagé dans les différentes variables du programme, et non les contraintes associées ; aussi, la méthode peut résulter en des faux positifs, c'est-à-dire trouver des vulnérabilités là où il n'y en a pas. Cette méthode permet de tester des modèles de faute très génériques, permettant donc parfois d'arriver à des preuves de sécurité très fortes. Mais dans les cas où la preuve ne peut aboutir, cette même généricité, couplée à la présence de faux positifs, permet difficilement d'évaluer la faisabilité d'une attaque. Or, le domaine de la sécurité est avant tout fait de compromis : il s'agit de protéger un système au plus juste coût. Des menaces mal définies peuvent conduire à un sous-dimensionnement ou un sur-dimensionnement des contremesures ; c'est pourquoi une modélisation précise est importante.

II.3.3. Contremesures logicielles

Les contremesures logicielles reposent globalement sur le même principe que les contremesures matérielles : ajouter de la redondance à l'application exécutée. Elles peuvent s'appliquer au niveau du code source ou au niveau assembleur, suivant le(s) modèle(s) de faute logiciel(s) considéré(s). Ces contremesures se répartissent en général en deux catégories : celles qui protègent le flot de données et celles qui protègent le flot de contrôle. A cela peuvent s'ajouter quelques techniques de programmation plus générales.

II.3.3.1. Contremesures sur le flot de données

Le premier type de contremesures vise à sécuriser le flot de données, c'est-à-dire s'assurer que des attaques ne puissent pas corrompre les résultats des opérations effectuées au sein du processeur. Dans cette catégorie, certaines contremesures sont similaires à ce qui se fait du côté matériel, avec par exemple la réplication du programme.

Duplication simple

Dans sa thèse [48], Moro propose, après avoir déterminé que son processeur était sensible spécifiquement aux sauts d'instruction, une contremesure basée sur la duplication d'instructions pour s'assurer que le programme donne toujours le bon résultat même en étant attaqué. Cette contremesure est basée sur le principe d'*idempotence*. Une instruction est dite idempotente si l'exécuter plusieurs fois de suite donne le même résultat qu'en l'exécutant une seule fois. Par exemple, l'instruction `A = B + 1` est idempotente, contrairement à l'instruction `A = A + 1` car cette dernière donnera à chaque fois un résultat incrémenté de 1.

Dans la contremesure développée par Moro, chaque instruction idempotente est simplement dupliquée ; ainsi, sauter l'exécution d'une des deux itérations n'a pas d'importance puisque l'autre itération calcule le bon résultat. Pour les instructions non idempotentes, Moro et al. proposent au préalable de les remplacer par une section de code qui aboutit au même résultat, mais dont chaque instruction est idempotente. Les Figure 3 et Figure 4 illustrent cette contremesure pour les deux instructions évoquées précédemment.

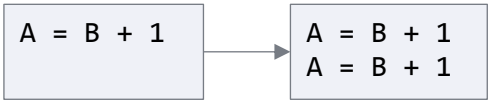

FIGURE 3: CONTREMESURE DE MORO SUR UNE INSTRUCTION IDEMPOTENTE

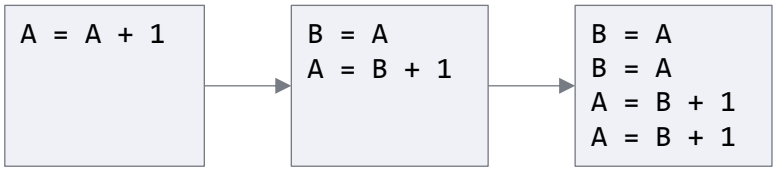

FIGURE 4 : CONTREMESURE DE MORO SUR UNE INSTRUCTION NON IDEMPOTENTE

Après avoir appliqué leur contremesure sur plusieurs applications, Moro et al. rapportent un surcoût qui s'élève en général entre 100% et 150% pour le nombre de cycles d'exécution, et entre 150% et 200% pour la taille du code. Si ces valeurs sont élevées, elles restent néanmoins comparables au surcoût d'autres contremesures basées sur la duplication.

Avec cette contremesure, il est possible de s'assurer qu'un programme s'exécute correctement même en présence d'un saut d'instruction. Moro et al. prouvent formellement cette conclusion dans [49]. Cependant, cette contremesure ne protège que contre les sauts d'instruction. Toute autre faute qui impacte le résultat d'une instruction (comme une corruption de registre) ne sera pas détectée.

Ainsi, Moro propose une contremesure aux propriétés intéressantes (le but n'est pas de détecter une faute mais de s'assurer du bon fonctionnement du programme même en présence d'une faute), mais avec un coût relativement élevé compte tenu du fait qu'elle ne peut protéger que contre un modèle de faute spécifique.

Duplication-comparaison

Une contremesure un peu plus générale est celle de la duplication-comparaison. Son rôle est de détecter une faute plutôt que de chercher à la corriger, en comparant les résultats de deux exécutions ; comme pour la duplication matérielle.

Ce processus de duplication peut s'effectuer au niveau de l'algorithme (exécution complète de l'algorithme, puis nouvelle exécution) ou au niveau des instructions (chaque instruction est exécutée plusieurs fois de suite). Comme le souligne Barenghi et al. dans [5], il est cependant plus facile pour un attaquant d'injecter la même faute dans deux exécutions successives d'un même programme que dans deux instructions qui se suivent. Une réplication au niveau des instructions semble donc une meilleure option puisqu'elle suppose d'avoir des moyens d'attaque plus avancés.

En ce qui concerne la duplication au niveau algorithmique, certains papiers recommandent également de diversifier l'implémentation de l'algorithme : au lieu d'exécuter deux fois strictement

les mêmes opérations, exécuter deux implémentations différentes de l'algorithme (mais qui mènent au même résultat). Une variante est d'utiliser un programme effectuant l'opération inverse, par exemple chiffrer un message, puis le déchiffrer, en vérifiant que le message obtenu est bien l'original. Ce type de contremesure est intéressant puisqu'il contre un problème évoqué précédemment, à savoir qu'injecter la même faute dans plusieurs exécutions d'un même programme n'est pas une hypothèse très forte sur le pouvoir de l'attaquant. Du point de vue de l'attaquant, il peut être bien plus complexe techniquement de provoquer les mêmes effets dans deux implémentations différentes d'un même algorithme. Cette solution nécessite cependant plus de temps de développement et n'est pas forcément utilisable dans tous les contextes.

Triplication

Il est également possible d'utiliser la triplication au niveau logiciel [5]. Comme en matériel, le but est de corriger une faute ou détecter deux fautes différentes. Il est à noter que l'augmentation de la taille du programme peut vite devenir problématique avec cette solution car l'organisation du vote de majorité devient plus complexe.

II.3.3.2. Contremesures sur le flot de contrôle

Outre les contremesures qui visent à protéger les données du programme, il existe aussi des contremesures qui cherchent à s'assurer de l'intégrité du flot de contrôle. Le flot de contrôle d'un programme est typiquement représenté sous forme d'un graphe avec des blocs, appelés blocs de base, et des liens entre ces blocs. Un bloc de base est une séquence d'instructions consécutives qui ne contient aucun saut ou destination d'un saut, hormis la dernière et la première instruction, respectivement. Le seul point d'entrée d'un bloc de base est sa première instruction et son seul point de sortie est sa dernière instruction. L'enchainement possible des blocs est représenté par des liens entre ces blocs. La Figure 5 présente un exemple de graphe de flot de contrôle.

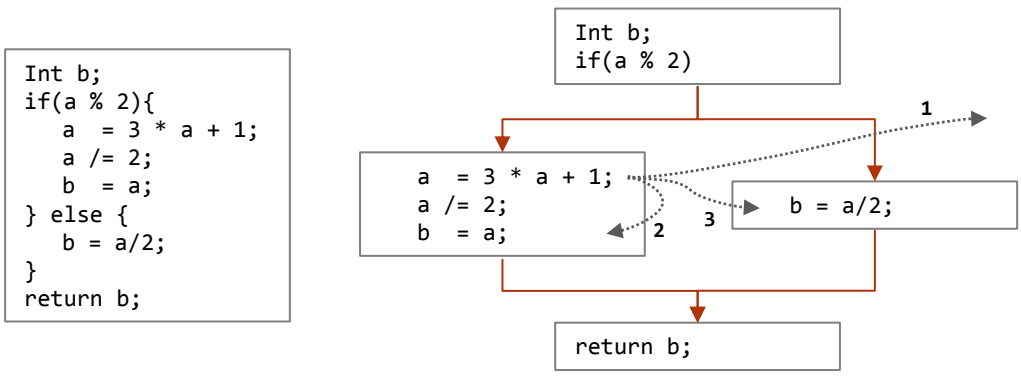

FIGURE 5 : EXEMPLE DE PROGRAMME ET DE SON GRAPHE DE FLOT DE CONTROLE. LES FLECHES EN POINTILLE REPRESENTENT LES POSSIBLES ERREURS PROVOQUEES PAR INJECTION DE FAUTE.

Le principe des contremesures d'intégrité de flot de contrôle est de s'assurer que le programme s'exécute suivant son graphe, c'est-à-dire qu'un attaquant ne puisse suivre un chemin non autorisé. Plusieurs menaces peuvent être répertoriées.

Une faute pourrait par exemple avoir pour effet de sauter vers un espace mémoire en dehors du programme (trajectoire 1 sur la figure). Ce type d'erreur est relativement difficile à contrer au niveau logiciel car il dépend d'instructions en dehors du programme. Les espaces mémoire non utilisés pourraient par exemple contenir des instructions qui redirigent vers une fonction de détection des erreurs.

Une autre conséquence possible d'une faute est de sauter des instructions dans un même bloc, comme sur la trajectoire 2 de la figure. Ce type de comportement est connu sous le nom de saut intra-bloc. Pour protéger le programme face à ce type d'erreur, l'une des solutions envisageables est d'utiliser un compteur qui s'incrémente à chaque instruction, puis de vérifier à la fin du bloc que le compteur a bien la valeur attendue. Cette solution est utilisée par exemple dans l'article [50]. Protéger contre les sauts intra-blocs est difficile au niveau logiciel : il faut ajouter des instructions dans un bloc de base pour s'assurer que chaque instruction de ce bloc est effectuée dans l'ordre… La solution du compteur n'est pas parfaite mais il existe peu d'alternatives. Pour la protection d'un système à un tel niveau de granularité, une contremesure matérielle du type *watchdog* est probablement plus pertinente.

Enfin, la troisième menace est celle des sauts inter-blocs. Ceux-ci consistent à effectuer un saut d'un bloc de base vers une instruction d'un autre bloc de base (trajectoire 3 sur la Figure 5). Pour éviter ce genre de comportements, de nombreux schémas de protection, basés sur la vérification de signatures, ont été développés. Une signature est assignée à chaque bloc de base lors de la compilation du programme. Pendant l'exécution, une signature est calculée à chaque transition vers un nouveau bloc, puis comparée à la signature de référence du bloc. Une incompatibilité dans les signatures signifie qu'un mauvais saut a été effectué. Les instructions de mise à jour et de comparaison de signatures sont typiquement placées au début ou à la fin des blocs (voire aux deux endroits). Les schémas CFCSS [51], YACCA [29] et SEDSR [1] sont des exemples de telles contremesures. Par ailleurs, le papier [64] donne une vue d'ensemble de ces différents schémas, leurs avantages et inconvénients, notamment en ce qui concerne le taux de détection et l'impact sur les performances du programme ; et les caractéristiques qui font un schéma de signature efficace.

II.3.3.3. Techniques de programmation

En plus des contremesures déjà citées, il existe au niveau logiciel certaines techniques de programmation qui peuvent permettre d'éviter des vulnérabilités. Dureuil et al. introduisent dans [21] plusieurs exemples.

Il est par exemple préférable d'utiliser des booléens durcis plutôt que de simples booléens. En effet, l'implémentation usuelle des valeurs booléennes est d'utiliser la valeur 0 pour représenter FAUX et n'importe quelle autre valeur pour représenter VRAI. Cette implémentation pose plusieurs problèmes. D'une part, il est relativement aisé de faire passer un booléen FAUX (0) à un booléen VRAI (valeur quelconque différente de 0) ; pour cela il suffit d'injecter n'importe quelle faute qui modifie sa valeur. D'autre part, même si au premier abord, il semble difficile de passer d'un booléen VRAI à un booléen FAUX (il faut arriver à une valeur spécifique), cela n'est pas forcément le cas en réalité. En effet, nous avons vu dans la partie modélisation que parmi les modèles de fautes usuels, le passage d'une variable à 0 est assez courant ; c'est un comportement que l'on retrouve fréquemment. Ainsi donc, avec l'implémentation des booléens par défaut, il semble relativement facile d'injecter des fautes pour faire basculer un booléen de FAUX à VRAI, ou de VRAI à FAUX. Pour parer à cette éventualité, l'idée est d'utiliser des booléens durcis, c'est-à-dire des booléens dont les deux valeurs sont fixées à des valeurs différentes de 0. Dans leur papier, Dureuil et al. proposent les valeurs 0x55 et 0xAA pour FAUX et VRAI, respectivement. Ces valeurs ont du sens en inspectant leur écriture binaire : 01010101 et 10101010. Il semble qu'avec ces valeurs il soit très compliqué de passer d'une valeur à l'autre puisque cela supposerait un attaquant capable d'injecter huit fautes dans un même registre, chaque faute provoquant un bit-flip dans le sens inverse des bits adjacents.

Avec cette implémentation, un booléen peut être soit FAUX (0x55), soit VRAI (0xAA), soit contenir une valeur incorrecte (toute autre valeur différente de 0x55 et 0xAA). Il faut donc aussi faire attention à adapter les conditions pour que les parties sensibles du programme ne puissent être accédées qu'avec une valeur correcte de booléen : tester si une variable est VRAIE n'est pas équivalent à tester si elle n'est pas FAUSSE.

Un autre exemple de technique de programmation est de tester le nombre d'itérations d'une boucle à la fin de cette boucle, pour vérifier que cette boucle ne s'est pas terminée prématurément à cause d'une faute. Un dernier exemple de technique de programmation est de supprimer les appels aux fonctions de sécurité en utilisant l'*expansion en ligne*, c'est à-dire en remplaçant l'appel des fonctions directement par leur code. Cela permet d'éviter des attaques qui permettraient de tout simplement sauter l'appel à ces fonctions.

II.3.3.4. Discussion sur les contremesures logicielles

Les contremesures logicielles ont l'avantage d'être plus faciles à mettre en place qu'au niveau matériel et sont également plus flexibles dans leur utilisation. Elles peuvent en particulier n'être appliquées qu'à certaines parties du programme pour diminuer leur coût, là où les contremesures matérielles ont un coût plus global. Pour finir, les contremesures logicielles peuvent être mises à jour si de nouvelles vulnérabilités sont trouvées, contrairement aux contremesures matérielles qui sont figées une fois le processeur fabriqué.

Cependant, ces contremesures reposent sur des modèles de faute logiciels qui ne peuvent représenter qu'imparfaitement les effets des fautes au niveau matériel. Dans [63], les auteurs évaluent la protection de 19 types de contremesures logicielles (qui sont plus ou moins des variations des contremesures présentées dans cette section) face à des injections simple-bit. Quel que soit le schéma de protection adopté, il existe à chaque fois des cas de fautes non détectées. Des modèles de faute logiciels trop peu représentatifs de situations réelles peuvent conduire à la conception de contremesures peu pertinentes. Ces contremesures sont efficaces pour contrer une menace bien circonscrite ; or les effets réels des fautes peuvent sortir des modèles de faute logiciels usuels.

II.4. Problématique

Dans la partie précédente a été esquissé le rôle central des modèles de faute logiciels dans la conception de mesures de protection efficaces face aux attaques en faute. Nous discutons maintenant ce point plus en détail.

II.4.1. Précision des modèles de faute logiciels

Les contremesures **logicielles** sont bâties pour contrer des modèles de faute **logiciels.** La modélisation des fautes à ce niveau est donc au centre du processus de sécurisation d'un système. Or, les modèles de faute logiciels utilisés typiquement peuvent être remis en question.

II.4.1.1. Evaluation de la précision de modèles de faute logiciels

Plusieurs travaux ont, par le passé, évalué les divergences entre modèles de faute logiciels et impacts réels des fautes dans les processeurs.

Dans [15], Cho et al. proposent une analyse quantitative des imprécisions obtenues en comparant les effets d'injections à différents niveaux d'abstraction. Pour cela, ils réalisent d'importantes campagnes d'injection par émulation d'un processeur d'architecture LEON3 exécutant différentes applications de la suite de benchmark SPECINT2000. Les injections de référence sont des injections

simple-bit de bit-flips à un cycle d'exécution choisi aléatoirement et dans une bascule choisie aléatoirement, en excluant les structures de type SRAM comme le banc de registres ou la mémoire cache. La raison invoquée pour cette restriction est que ces structures sont généralement protégées, notamment par des codes correcteurs d'erreur.

Les injections de référence sont comparées à deux autres types d'injection à des niveaux d'abstraction supérieurs : d'une part, des injections simple-bit dans les registres visibles depuis le côté logiciel (c'est-à-dire les registres manipulés au niveau assembleur), et d'autre part des injections simple-bit dans les variables du programme, donc au niveau du code source.

Pour chaque injection, les résultats de l'expérience sont classés dans cinq catégories :

- Aucun effet par rapport à une exécution non fautée
- Effet visible sur les sorties du programme
- Aucun effet visible sur les sorties du programme, mais le processeur finit dans un état différent par rapport à une exécution non fautée (la faute a donc bien un effet sur l'architecture, mais qui ne s'est pas manifesté de manière visible sur les sorties du programme)
- Problème d'exécution (division par zéro, instruction invalide, etc)
- Non terminaison du programme

La comparaison des résultats des injections aux différents niveaux d'abstraction montre de manière assez éloquente les disparités qui existent entre les différents niveaux. Le nombre de résultats dans chacune des cinq catégories varie grandement suivant le niveau d'abstraction considéré, parfois avec un ordre de grandeur de décalage. Cette première conclusion permet déjà de montrer les problèmes de modélisation qui existent actuellement : les modèles de haut niveau donnent, statistiquement, des résultats bien différents d'injections au niveau RTL. Bien sûr, ce résultat ne prend en compte que quelques modèles de faute logiciels, et pas, par exemple, de modèles agissant sur le flot de contrôle, comme le saut d'instruction. Il représente malgré cela un résultat intéressant, dans la mesure où, à notre connaissance, aucune étude quantitative de ce genre ne prend en compte tous les modèles de faute logiciels. Il est par ailleurs difficile de définir ce que l'on entend par « tous les modèles », au vu de leur diversité.

II.4.1.2. Origine des imprécisions

Dans le même papier, les auteurs étudient également la propagation des fautes injectées dans les bascules en contrôlant les signaux envoyés à différentes structures du processeur. Les fautes peuvent être masquées (aucune propagation visible au niveau logiciel), se propager dans le banc de registres, se propager dans la mémoire, se propager dans une structure annexe (comme le compteur de programme, le contrôleur d'interruption ou les signaux de contrôle du cache), ou se propager dans plusieurs de ces catégories à la fois (exemple : impacter à la fois le banc de registres et une structure annexe). Les auteurs montrent ainsi que seules 14,4% des fautes non masquées se propagent uniquement dans le banc de registres, la mémoire, ou une combinaison des deux. Si l'on s'en tient à ces résultats, les modèles de corruption de registre et de corruption de mémoire semblent bien peu représentatifs. 54,8% des fautes non masquées ne se propagent que dans des structures annexes (dont environ un tiers dans le compteur de programme). Or, ce type de faute est à l'origine de 21,91% des injections menant à des effets visibles sur la sortie du programme. Cela montre d'une part que les fautes peuvent avoir des comportements complexes dans l'architecture, pas forcément représentés dans les modèles de faute logiciels ; et d'autre part que ces comportements peuvent

aboutir à des effets bien visibles. Tous ces résultats sont bien sûr à mettre en perspective : ils résultent d'une étude sur un processeur spécifique, avec une certaine méthode d'injection, etc.

Toutefois, l'impact de l'architecture du processeur sur les comportements observés est un fait également noté dans d'autres travaux. Dans [65], Wang et al. étudient l'impact de fautes injectées dans différentes structures d'un processeur superscalaire. Les auteurs mettent en avant le fait que chaque structure conduit à des comportements différents. Cela permet de prendre conscience de cette diversité des effets des fautes induites par la microarchitecture, mais les effets ne sont pas caractérisés précisément ; il s'agit surtout d'une étude quantitative.

Dans un article de Yuce et al. [67], au titre quelque peu provocateur (« La résistance logicielle aux fautes est futile »), les auteurs montrent comment de simples perturbations de l'horloge peuvent contrecarrer certaines contremesures logicielles. La raison en est la suivante. Les perturbations de l'horloge ont un impact global sur le processeur, donc ils peuvent toucher de multiples structures en même temps. Or la quasi-totalité des processeurs actuels possèdent un pipeline, ce qui signifie que plusieurs instructions sont en cours d'exécution en même temps. Donc une attaque par perturbation peut facilement impacter plusieurs instructions consécutives, ce que ne prévoient pas (ou rarement) les modèles de faute logiciels typiques. C'est pourquoi les contremesures bâties contre ces modèles ne sont pas efficaces.

Dans [8], les auteurs mènent une étude pour déterminer au niveau logiciel quels sont les registres les plus vulnérables à des injections de faute. Ils s'aperçoivent cependant, en examinant leurs résultats, que les fautes directes dans les registres assembleur n'ont pas autant d'impact que prévu. En effet, certaines structures internes au processeur font que ces registres ne sont pas autant utilisés que le code assembleur laisserait penser. En l'occurrence, une optimisation très commune, appelée « forwarding » - que l'on étudiera plus en détail dans la suite de cet exposé - se déclenche fréquemment pour contourner le banc de registre afin de gagner en performances. Cet article remet donc en cause la pertinence du modèle de corruption de registre. Il illustre les difficultés de modélisation liées aux diverses optimisations présentes dans les processeurs actuels, et en particulier, les limites d'une modélisation trop portée sur une vision simpliste de l'architecture d'un processeur. Le modèle de corruption de registre, qui semble aller de soi eu égard à une architecture simpliste, n'est pas nécessairement pertinent pour une architecture réelle.

Ces différents travaux permettent d'une part de se rendre compte des imprécisions des modèles de faute logiciels typiques, et d'autre part d'esquisser la cause de ces imprécisions. Les méthodes d'analyse au niveau logiciel buttent sur la réalité de la complexité des processeurs actuels.

II.4.2. Réflexions sur l'origine des imprécisions

II.4.2.1. Rôle de l'architecture du processeur

Lorsque nous nous sommes intéressés à la modélisation des fautes au niveau matériel, en section II.2.1, nous avons pu remarquer que les différents niveaux d'abstraction avaient un lien logique entre eux. Dans le cas d'injections laser, nous avons vu qu'une modélisation au niveau RTL découle assez naturellement des effets induits au niveau physique, c'est-à-dire avec la création de paires électrons-trous. Ce fil de conséquences logiques est quelque peu brisé lorsque nous en venons à considérer le niveau logiciel. Il y a en effet un écart considérable entre les niveaux RTL et logiciel. Comment un modèle de faute RTL de type bit-flip ou bit-set peut-il se transformer en des modèles logiciels comme le saut d'instruction ou la corruption de registres ?

Le lien n'est pas aussi évident que le passage d'une modélisation physique à une modélisation RTL. Il faut, pour expliquer les impacts au niveau logiciel, faire intervenir l'architecture du processeur sur lequel est injectée la faute. C'est en effet l'architecture du processeur qui permet à un ou plusieurs bit-flips de se propager et de perturber la bonne exécution d'un programme, ainsi que l'illustre la Figure 6. La question de la modélisation des fautes au niveau logiciel revient donc à une étude de l'architecture des processeurs. C'est là la clé d'une bonne prise en compte des effets des fautes au niveau logiciel.

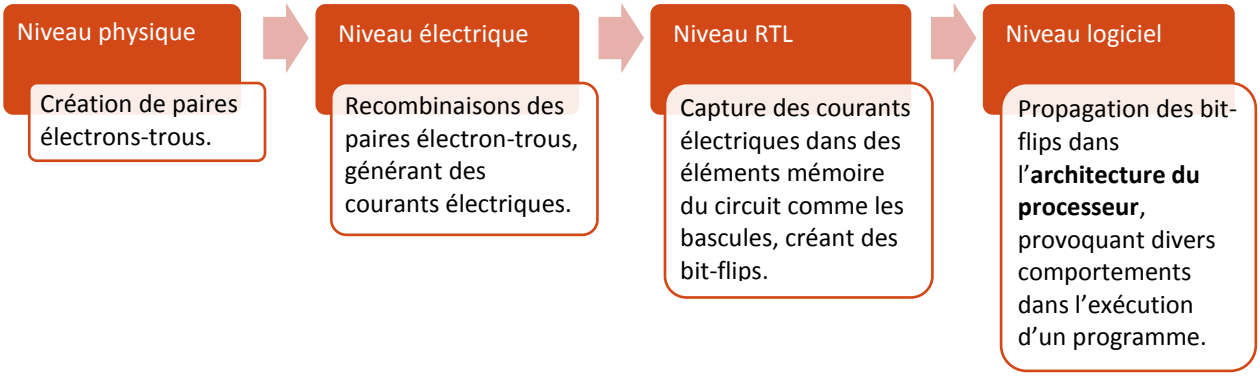

FIGURE 6 : NIVEAUX D'ABSTRACTION POUR LA MODELISATION D'ATTAQUES LASER

II.4.2.2. Complexité des processeurs

Les modèles de faute logiciels les plus utilisés ont du sens lorsque nous nous référons aux éléments basiques de l'architecture d'un processeur. Un processeur, quel que soit son architecture, a des emplacements mémoire dont la valeur pourra être corrompue. Tous les processeurs ont également un Compteur de Programme (PC), et injecter une faute dans ce compteur a des chances de provoquer un saut d'instruction. Cependant, un processeur ne se limite pas à ces structures. L'idée qu'un processeur est constitué essentiellement d'une Unité Arithmétique et Logique (ALU), une mémoire et un compteur de programme est erronée ; elle ne correspond pas, ou en tout cas plus, à la réalité des architectures actuelles. Depuis leur apparition dans les années 1970, les micro-processeurs ont beaucoup évolué ; et notamment, de nombreuses structures d'optimisation ont été ajoutées. Ces évolutions ont progressivement entrainé une complexification des architectures et de leurs implémentations, qui se sont par conséquent éloignées de l'image du processeur simple.
Si les modèles de faute logiciels usuels s'appliquent particulièrement bien à une vision simplifiée de l'architecture d'un processeur, leur pertinence peut être remise en question lorsqu'on examine plus précisément les spécificités de ces mêmes architectures. Si un modèle de corruption de registre ou de saut d'instruction est facile à appréhender dans un processeur simple, qu'en est-il dans un processeur avec un pipeline, ou dans un processeur à exécution dans le désordre ?

II.4.2.3. Diversité des architectures et de leurs implémentations

En parallèle de cette inadéquation entre simplicité des modèles logiciels et complexité des processeurs actuels, il existe une deuxième pierre d'achoppement pour expliquer les disparités entre modèles logiciels et effets réels. En général, les modèles de faute logiciels sont utilisés pour évaluer la robustesse de divers programmes, **indépendamment** du processeur sur lequel ces programmes seront exécutés. Or il semblerait étonnant que des processeurs d'architecture différente, se

comportent de la même manière lorsqu'ils sont attaqués. Pourquoi, après l'injection d'une faute, un processeur d'architecture RISC se comporterait-il de la même manière qu'un processeur d'architecture CISC ? Pourquoi, après l'injection d'une faute, un processeur Intel se comporterait-il de la même manière qu'un processeur ARM ? Pourquoi, après l'injection d'une faute, un processeur super-scalaire avec exécution spéculative et pipeline de six étages se comporterait-il de la même manière qu'un processeur non pipeliné ou qu'un processeur de mots d'instruction très longs (VLIW) ? En appliquant des modèles de faute logiciels trop génériques, des effets spécifiques, conséquence de la diversité des processeurs, sont largement ignorés.

Dans [16], l'auteur quantifie les différences d'effets des fautes sur divers programmes, pour des processeurs différents. Il montre ainsi que les effets obtenus sur une architecture RISC-V sont différents de ceux obtenus sur une architecture LEON3. Il faut noter également qu'il conclut qu'il existe une forte corrélation des résultats entre deux implémentations très différentes (l'une utilisation l'exécution dans l'ordre, l'autre dans le désordre) d'une même architecture (en l'occurrence, RISC-V). Ce résultat semble invalider ce qui a été dit dans cette section, mais il faut cependant nuancer cette affirmation : l'étude en question est purement quantitative ; elle ne s'intéresse qu'au taux d'erreur obtenu dans différentes applications. Il n'y a pas de différenciation dans les effets précis qu'a eu la faute sur l'exécution (la seule différenciation est sur l'issue de l'exécution, répartie en quatre catégories : erreur, exception, aucun effet, ou non-terminaison du programme). Or, un modèle de faute logiciel est avant tout fait pour représenter l'effet d'une faute, de manière qualitative.

II.4.2.4. Représentation des imprécisions

Les modèles de faute logiciels ne sont qu'une abstraction des effets réels d'une faute au niveau matériel. Ce processus d'abstraction des effets s'accompagne d'une perte de précision vis-à-vis des effets réels. Le diagramme de la Figure 7 illustre cela. Il représente d'une part l'espace des effets réels des fautes, et d'autre part l'espace des effets prédits par les modèles de faute logiciels. Sur cette figure apparaissent trois zones. Celle du centre correspond aux effets réels qui sont correctement représentés par les modèles de faute logiciels. Le but du processus de modélisation est de rendre cette zone aussi large que possible. Cependant, les approximations des modèles mènent à deux types de problèmes. D'une part, tous les effets réels ne sont pas prédits par les modèles de fautes ; cela correspond à la partie gauche du diagramme. Il s'ensuit que les contremesures logicielles ne protègent pas forcément face à ces attaques. D'autre part, les effets prédits par les modèles logiciels ne correspondent pas toujours exactement aux effets réels ; ces modèles peuvent prédire des attaques qui ne sont pas réalisables en réalité. Se prémunir de ces attaques est donc inutile et les contremesures associées ne résultent qu'en une perte de performances et un accroissement des coûts de développement et de fabrication.

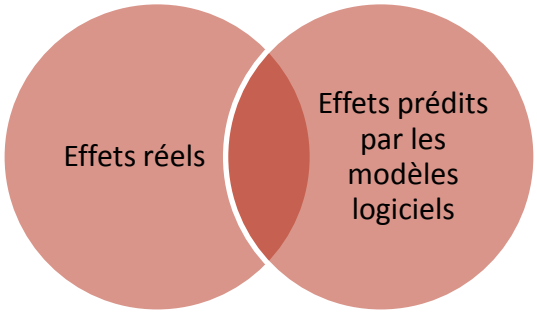

FIGURE 7: DIAGRAMME DES EFFETS REELS ET PREDITS

II.4.2.5. Conclusion

Les modèles de faute logiciels usuels décrivent des effets dont on espère qu'ils correspondent aux effets réels induits par les fautes. Ces modèles sont bâtis suivant une représentation simple des processeurs, qui, bien que correspondant toujours grossièrement aux architectures actuelles, laissent de côté la complexité et la diversité réelles des processeurs. La complexité entraîne une difficulté à modéliser correctement les fautes et la diversité implique que chaque processeur doit avoir ses propres modèles de faute logiciels. Cette difficulté dans la modélisation entraîne des imprécisions qui peuvent fausser les analyses de sécurité et qui peuvent conduire à une surprotection, ou, plus grave, une sous-protection des systèmes.

Ainsi, l'architecture des processeurs crée une inadéquation entre modèles logiciels et réalité. Une question demeure alors : comment lutter contre ces imprécisions ? Comment rendre les modèles de faute logiciels plus représentatifs ?

II.4.3. Construction de modèles à partir d'injections expérimentales

Pour répondre à ces questions, plusieurs travaux ont déjà été menés pour prendre en compte les spécificités des processeurs lors de la modélisation des fautes au niveau logiciel.

Balasch et al. étudient dans leurs travaux [2] la modélisation d'injections par perturbation de l'horloge, sur un microcontrôleur AVR. Les attaques sont menées sur différents types d'instructions de l'architecture AVR. Plusieurs effets complexes sont observés, dont notamment des remplacements d'instructions, ou encore, lors d'une opération mémoire, le chargement de la dernière valeur écrite/chargée au lieu de la donnée voulue. Les auteurs arrivent toutefois à la conclusion que beaucoup d'effets sont trop complexes pour pouvoir être interprétés correctement. Pour expliquer les résultats obtenus, les auteurs invoquent la complexité d'une architecture Harvard face à une architecture Von Neumann, avec notamment l'utilisation d'un pipeline.

Moro propose dans sa thèse [48] une caractérisation des effets induits par des injections électromagnétiques sur un processeur ARM. Les injections sont réalisées pendant l'exécution de programmes simples, comme une suite d'instructions NOP (*no operation* ; instruction qui ne fait rien), une lecture mémoire isolée, ou encore l'addition d'éléments d'un tableau. Après plusieurs expérimentations, Moro en arrive à la conclusion que les fautes s'assimilent à des remplacements d'instruction ou à la corruption de valeurs lues en mémoire. Il explique également qu'un modèle de saut d'instruction permet d'expliquer 25% des effets obtenus lors des campagnes d'injection. Il développe par la suite une contremesure, présentée en section II.3.3.1, pour protéger face à cette menace. Dans ce travail, c'est donc la caractérisation des effets des fautes sur un processeur spécifique qui a guidé la protection du système.

Kelly et al. caractérisent les effets obtenus par injection laser dans un microcontrôleur AVR [35]. Divers programmes étaient utilisés pendant les injections, comme des séquences de NOP, des opérations arithmétiques, des accès mémoire ou des opérations de branchement. Parmi les effets observés, des sauts d'instruction ou encore des corruptions de mémoire ou de registre (même si ces cas ne sont pas détaillés).

Dans sa thèse [20], Dureuil propose une approche d'inférence de modèles de faute logiciels pour étudier les effets provoqués par des injections électromagnétiques sur une carte spécifique. Le problème est posé de la façon suivante. Le but est de caractériser les effets provoqués par des injections électromagnétiques sur une carte électronique. La carte est considérée comme une boîte

noire : l'architecture interne n'est pas connue et l'expérimentateur ne peut qu'inférer un modèle de faute à partir d'expérimentations réelles. Entrent alors en jeu deux types de paramètres. D'une part les paramètres d'équipement, qui sont répartis en trois catégories : dimension spatiale (coordonnées (x,y) de l'injection), dimension temporelle (instant et durée d'injection) et dimension énergétique (intensité de l'injection). D'autre part, les paramètres de code, avec également une dimension spatiale (emplacement mémoire), une dimension temporelle (moment de l'injection) et une dimension qualitative qui donne l'effet de la faute. Quatre types d'effets sont considérés : ajout/modification de lecture/écriture (que ce soit dans la mémoire ou les registres). La méthode d'inférence consiste à relier les paramètres d'équipement aux paramètres de code en réalisant des expériences et, itérativement, à arriver à une solution où chaque valeur des paramètres d'équipement est associée à un modèle de faute logiciel. Divers programmes de détection sont utilisés afin de réduire la dépendance des modèles à ces programmes. Cette approche possède deux aspects particulièrement intéressants. Le premier est de proposer une méthode de modélisation pour un processeur donné : le but n'est pas d'arriver à un modèle de faute général, mais de proposer un panel de comportements possibles spécifiquement sur le processeur étudié. Le second aspect intéressant est une volonté non pas seulement de modéliser les fautes au niveau logiciel, mais plutôt de lier des modèles logiciels spécifiques à des paramètres d'injection expérimentale. Le passage du côté logiciel vers le côté matériel reste possible, ce qui peut permettre de vérifier expérimentalement des vulnérabilités trouvées au niveau logiciel.

Proy et al. présentent quant à eux une méthodologie de caractérisation des effets d'injections électromagnétiques sur un processeur ARM superscalaire à exécution dans le désordre [58]. Les injections sont réalisées pendant l'exécution de plusieurs programmes simples, comme une séquence d'instructions NOP, ou une séquence d'incrémentation d'un ou plusieurs compteurs. Ces expériences permettent de mettre en avant différents modèles logiciels, comme le saut d'instruction ou la corruption de registre. Dans ce second cas, la valeur corrompue provient fréquemment d'une instruction précédemment exécutée, et les auteurs invoquent, pour expliquer ce phénomène, le rôle de certaines structures d'optimisation du processeur. Un des éléments intéressants de ce travail est de considérer très justement qu'un processeur superscalaire à exécution dans le désordre a une architecture bien différente d'un processeur simple, et que cette différence peut entraîner des effets plus complexes.

Ces différents travaux se basent sur des résultats expérimentaux, avec un processeur considéré comme une « boîte noire ». Le fonctionnement interne du processeur n'est pas connu des expérimentateurs et les modèles ne peuvent donc qu'être inférés à partir des effets visibles des fautes. L'inférence de modèles de faute logiciels à partir de résultats expérimentaux nécessite donc un a priori sur le fonctionnement interne du processeur. Sans plus de connaissances sur son implémentation réelle, l'expérimentateur ne peut que se reposer sur des considérations théoriques quant à l'architecture du processeur. Ces procédés se heurtent donc toujours en partie aux limitations des modèles de fautes logiciels évoquées précédemment ; à savoir le décalage entre une vision simple du processeur et sa complexité réelle. Ils sont néanmoins la seule solution envisageable pour caractériser les modèles de fautes d'un processeur spécifique, dans les cas où l'implémentation de celui-ci n'est pas accessible.

Il est utile de noter ici que l'étape de modélisation des fautes est une procédure sensible à plusieurs paramètres, comme le programme exécuté lors des injections ou la méthode d'injection considérée. Si le programme exécuté est trop peu diversifié, le risque est de modéliser des effets qui n'arrivent

que dans des cas très particuliers. Pour lutter contre ce biais, les différents travaux présentés ici utilisaient plusieurs programmes dans leurs campagnes d'injection. Le second biais est celui de la méthode d'injection. Les modèles obtenus par injection électromagnétique sont-ils valables avec d'autres méthodes ? Restent-ils seulement valables en ne modifiant que l'antenne utilisée ? Il est malheureusement difficile de contrer cette influence ; chaque méthode d'injection, quelle qu'elle soit, introduit fatalement un biais dans les résultats.

II.4.4. Formulation de la problématique

Outre les processeurs fermés, utilisés dans les travaux précédents, il existe aussi des processeurs open-source, dont l'implémentation est accessible par tous. C'est le cas par exemple des processeurs d'architecture RISC-V, un projet lancé il y a une dizaine d'années par l'université de Californie. Cette architecture de processeur, que nous étudierons dans le chapitre suivant, est aujourd'hui utilisée par plusieurs entreprises, comme SiFive ou Western Digital, et il est probable que son importance grandisse dans les années à venir.

La connaissance de la description RTL d'un processeur permet d'aller plus loin dans la compréhension des effets provoqués par l'injection de faute. Elle permet de créer des modèles de faute logiciels qui prennent en compte les spécificités de l'architecture du processeur, c'est-à-dire de prendre en compte à la fois l'impact de la complexité et de la diversité des processeurs.

Avec la description RTL d'un processeur, la façon la plus naturelle d'injecter des fautes est la simulation ou l'émulation. Ces méthodes d'injection sont intéressantes pour deux raisons, déjà évoquées précédemment, à savoir qu'elles permettent un très bon contrôle et une bonne observabilité des injections, caractéristiques primordiales pour étudier précisément les comportements induits dans le processeur ; et qu'elles peuvent être pratiquées relativement tôt dans le flot de développement, permettant de limiter les coûts engendrés par une détection trop tardive de comportements problématiques.

Ces caractéristiques concourent à créer une méthode prometteuse, permettant une étude approfondie des effets logiciels induits par des injections de fautes au niveau RTL. Cette meilleure connaissance doit ensuite permettre d'analyser la sécurité de programmes plus efficacement. Le but de cette thèse est donc de développer une telle méthode.

Pour cela, l'étude est découpée en trois parties. La première vise tout d'abord à étudier « manuellement » la description RTL d'un processeur et d'explorer en quoi différentes structures de l'architecture peuvent mener à des comportements non modélisés par les modèles de faute logiciels usuels. De premières injections sont menées en simulation, afin d'étudier précisément leurs effets et leurs conséquences sur des contremesures logicielles typiques, ainsi que sur une application sécurisée.

Dans la seconde partie, un outil pour représenter ces fautes au niveau logiciel est développé ; puis, une approche de modélisation est définie, afin de réaliser des campagnes d'injection conséquentes et ainsi modéliser et valider de nombreux comportements. Dans cette approche, plusieurs métriques sont définies pour évaluer la pertinence des modèles de fautes logiciels. Cette approche vise également à conserver, à l'instar des travaux de Dureuil, le lien entre modèles logiciels et paramètres matériels.

Enfin, la troisième et dernière partie étudie la possibilité d'utiliser ces nouveaux modèles de faute logiciels dans des analyses statiques de programmes. Les méthodes de l'état de l'art étant souvent

liées à des modèles particuliers, il est intéressant de voir si nos modèles peuvent être utilisés dans des analyses de sécurité complexes.

Pour résumer en des termes plus simples, les trois chapitres de cette thèse répondent aux trois questions suivantes, à propos des modèles de faute prenant en compte l'architecture du processeur. Que sont-ils ? Comment les modéliser ? Comment les utiliser dans des analyses de sécurité ?

Chapitre III. Injection de faute dans l'architecture des processeurs

Résumé du chapitre

Dans ce chapitre, nous explorons en quoi l'architecture des processeurs modernes rend complexe la phase de modélisation des fautes au niveau logiciel. Le passage d'une modélisation RTL vers une modélisation logicielle s'accompagne de nombreux effets complexes que seule une analyse précise de l'architecture du processeur peut permettre de comprendre.

C'est cette analyse qui est menée dans ce chapitre, d'abord de manière théorique, puis de manière pratique en simulation RTL. Certaines optimisations du processeur, comme le pipeline, le *forwarding*, ou l'exécution spéculative sont particulièrement étudiées. Cette étude montre par exemple que l'architecture du processeur peut privilégier certaines valeurs, comme les valeurs utilisées précédemment dans l'exécution du programme ; ou encore rendre plus sensibles certains registres. Les comportements exposés peuvent remettre en question certaines contremesures logicielles classiques ainsi qu'une application de vérification de PIN sécurisée.

Sommaire

III.1. Introduction du chapitre

Pour aller au-delà des modèles de faute typiques exposés dans le chapitre précédent, il est nécessaire de mieux comprendre la propagation des fautes dans une architecture de processeur. Ce chapitre a un caractère fondateur dans cette thèse puisqu'il expose concrètement plusieurs mécanismes de propagation des fautes dans une architecture de processeur, montre la complexité de la modélisation de ces effets et illustre le potentiel de vulnérabilités qui en découle.

Nous commencerons par exposer quelques notions théoriques d'architecture des processeurs, présenterons quelques optimisations communes, et discuterons du rôle qu'elles peuvent jouer dans un contexte d'injection de fautes. Puis nous étudierons plus en détail l'implémentation de certaines structures dans un processeur d'architecture RISC-V. Enfin, ces structures seront attaquées en simulation RTL pour explorer les comportements fautifs qu'il est possible d'obtenir, montrer leur impact sur des contremesures logicielles typiques et sur une application de vérification de PIN sécurisée.

III.2. Introduction à l'architecture des processeurs

Comme exposé au chapitre précédent, une modélisation pertinente de l'effet des fautes au niveau logiciel passe par l'analyse de l'architecture d'un processeur. Pour pouvoir bien appréhender les comportements fautifs qui seront évoqués par la suite, il est donc intéressant de s'arrêter un moment pour passer en revue quelques notions importantes de l'architecture des processeurs. Cette section a également pour but de discuter de l'impact de certaines optimisations communes aux processeurs sur l'analyse de sécurité au niveau logiciel. Nous étudierons en particulier les notions de pipeline, de *forwarding* (défini plus loin) et d'exécution spéculative.

III.2.1. Eléments basiques d'une architecture

Partons d'une vue très simpliste de l'architecture d'un processeur, à laquelle nous adjoindrons certaines optimisations communes. Une représentation basique d'une architecture de processeur est exposée en Figure 8.

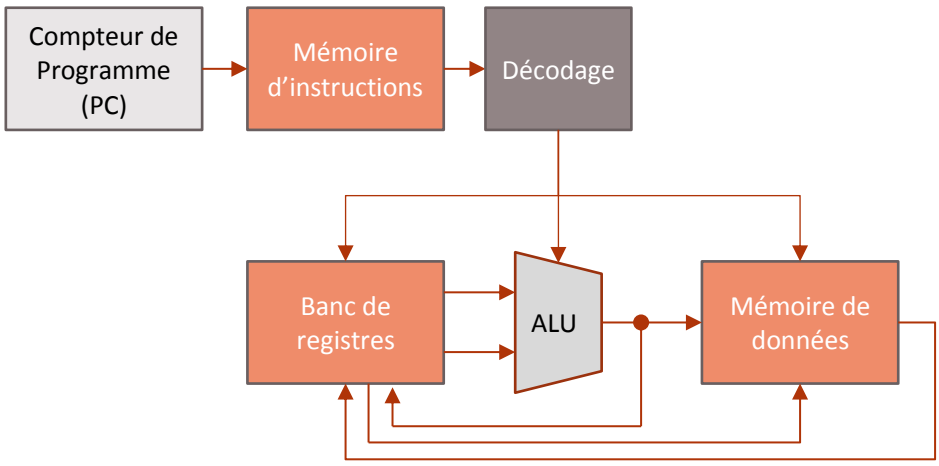

FIGURE 8 : REPRESENTATION BASIQUE D'UNE ARCHITECTURE DE PROCESSEUR

Sur cette figure, nous trouvons tout d'abord le Compteur de Programme (PC) qui permet de sélectionner dans la mémoire d'instructions la prochaine instruction à exécuter. Une fois cette instruction décodée, elle peut être exécutée : les arguments de l'instruction sont récupérés dans le banc de registres puis envoyés à l'Unité Arithmétique et Logique (ALU). Le résultat du calcul est ensuite écrit dans le banc de registres, ou peut servir d'adresse pour des opérations mémoire dans la mémoire de données. Ceci constitue un schéma simple que nombre de processeurs actuels suivent encore, avec bien sûr divers degrés de sophistication.

Sur cette figure, nous pouvons interpréter de manière assez simple les différents modèles de faute logiciels typiques évoqués dans le chapitre précédent. Un saut d'une ou plusieurs instruction(s) correspond typiquement à une faute dans le compteur de programme par exemple. Un modèle de remplacement d'instruction correspond à une faute dans la mémoire d'instructions ou dans la structure de décodage. Une corruption de registre correspond à une faute dans le banc de registre. De même, une corruption de mémoire correspond à une faute directement dans la mémoire de données. Pour finir, une erreur de calcul correspond à une faute dans l'ALU.

Cette correspondance entre les modèles de fautes typiques et le schéma simple de l'architecture d'un processeur n'est pas anodine. Comme ces modèles sont souvent construits sans véritable considération pour la réalité matérielle, ils ne peuvent que se référer à une vue très générale du processeur. Mais un processeur réel est plus complexe que cela.

III.2.2. Pipeline

Dans la plupart des processeurs actuels, les instructions ne sont pas exécutées *atomiquement*. En d'autres termes, elles ne sont pas exécutées en un seul cycle d'horloge, mais en plusieurs étapes. Typiquement, une instruction passe d'abord dans une phase de décodage de l'instruction, puis dans une phase de calcul et enfin dans une phase d'écriture des résultats. Ce découpage en différentes tâches permet de créer un *pipeline* où les instructions sont exécutées à la chaîne. Ces caractéristiques permettent d'augmenter les performances du processeur, pour deux raisons. La première est que la définition de tâches plus élémentaires permet d'augmenter la fréquence d'utilisation du processeur : les chemins critiques du processeur deviennent plus courts. La seconde est que cela permet d'exécuter plusieurs instructions en même temps : quand l'instruction 1 est dans l'étage d'écriture, l'instruction 2 est dans l'étage de calcul et l'instruction 3 dans l'étage de décodage.

Cette façon de faire a permis de beaucoup augmenter les performances des processeurs, au prix d'une complexité accrue. Cette première optimisation a déjà une conséquence vis-à-vis du point de vue logiciel. Côté logiciel, nous considérons que quand l'exécution du programme en est à l'instruction i, c'est que les instructions précédentes sont complètement terminées. Les instructions sont considérées comme atomiques : elles sont exécutées « instantanément ». Nous voyons donc apparaître ici une première divergence entre une vision logicielle de l'exécution d'une instruction et sa réalité matérielle. Cette divergence ne sera pas sans conséquence par la suite car elle introduit une complexité pour l'analyse de sécurité. Si le fait d'injecter une faute « dans une instruction » est facile à visualiser au niveau logiciel, qu'en est-il au niveau matériel étant donné que l'exécution de cette instruction s'étend sur plusieurs coups d'horloge ?

III.2.3. Structure de résolution des aléas

L'utilisation d'un pipeline n'est pas sans conséquence non plus sur l'exécution d'un programme. En effet, il est fréquent qu'une même variable soit utilisée dans plusieurs instructions consécutives. Dans ce cas apparaissent des problèmes de dépendances, appelés aléas de données : le pipeline est sensé exécuter plusieurs instructions en même temps, mais certaines ont besoin de résultats pas encore connus. Il existe plusieurs types de dépendances mais par souci de simplicité nous nous limiterons ici à expliquer le plus courant : la situation dans laquelle une première instruction fait un calcul et l'instruction suivante fait un calcul dont l'un des opérandes est le résultat de la première. Cette situation est représentée en Figure 9.

```
ADD a2 = a0 + a1
ADD a3 = a2 + a0
```

FIGURE 9: EXEMPLE DE DEPENDANCE ENTRE INSTRUCTIONS. LE RESULTAT DE LA PREMIERE INSTRUCTION DOIT ETRE IMMEDIATEMENT UTILISE PAR L'INSTRUCTION SUIVANTE.

Si l'on observe le Tableau 1 qui représente l'état du pipeline lors de l'exécution de cette séquence d'instructions, nous pouvons voir qu'au temps t_2, la seconde instruction ne peut continuer ; elle ne peut effectuer l'étape d'exécution car le résultat de l'instruction précédente n'est pas encore disponible (la première instruction n'est pas encore terminée).

TABLEAU 1 : EXEMPLE DE PROBLEME INTRODUIT PAR LE PIPELINE DANS LE CAS DE DEPENDANCES ENTRE INSTRUCTIONS

Temps	Décodage	Exécution	Ecriture des résultats
t_0	ADD **a2** = a0 + a1	-	-
t_1	ADD a3 = **a2** + a0	ADD **a2** = a0 + a1	-
t_2	?	?	ADD **a2** = a0 + a1

Pour débloquer cette situation, plusieurs solutions peuvent être envisagées. La première est de stopper temporairement l'exécution de la seconde instruction, le temps que la première se termine. Cette solution a bien sûr pour contrepartie une perte de performances. Comme les situations de dépendances entre instructions consécutives arrivent très fréquemment, cette contrepartie est assez pénalisante et annihile une partie des avantages créés par l'utilisation d'un pipeline.

Une seconde solution existe, qui porte le nom de *forwarding*. En français, le terme équivalent serait « envoi » ou « dérivation », mais ceux-ci étant assez peu utilisés, nous nous en tiendrons au terme anglais. Le *forwarding* consiste à ajouter des chemins dans le processeur pour pouvoir réinjecter une valeur du pipeline directement dans l'ALU, sans passer par le banc de registre. En d'autres termes, le résultat de la première instruction peut être directement réutilisé par l'ALU avant même que cette instruction soit terminée (i.e., avant même que le résultat soit écrit dans le banc de registres). Cette solution permet donc d'exécuter les instructions de la Figure 9 sans perdre un seul cycle. D'un point de vue performance, la solution du *forwarding* est bien plus avantageuse que la première solution, c'est pourquoi c'est celle qui est utilisée dans tous les processeurs de nos jours.

Cette solution est complètement transparente du point de vue logiciel. Pourtant, elle accroit encore la différence de vision entre niveau matériel et niveau logiciel, par l'introduction de ce concept : les résultats des instructions peuvent être utilisés avant même que ces instructions soient terminées.

Nous verrons par la suite que cette particularité est à l'origine de comportements fautifs atypiques au niveau logiciel.

Il est à noter qu'il existe une troisième manière de résoudre le problème de dépendances de variables, qui est de tout simplement d'éviter ces dépendances en exécutant les instructions dans un ordre différent. Cette réorganisation peut se faire en amont, en utilisant un compilateur qui réorganise le séquencement des instructions de manière à éviter les dépendances entre instructions consécutives, mais peut également se faire en vol dans le cas de processeurs avec des architectures dites à exécution dans le désordre. Ces processeurs ne sont cependant pas l'objet de cette thèse ; nous nous limiterons à un processeur plus simple qui exécute les instructions dans l'ordre où elles apparaissent. Nous pouvons cependant remarquer le fait que ne pas exécuter les instructions dans l'ordre est encore une optimisation qui élargit l'écart de vision entre la partie logicielle et la partie matérielle. L'analyse de sécurité sous cette hypothèse s'en trouverait encore plus complexe.

III.2.4. Exécution spéculative

L'utilisation d'un pipeline a une autre conséquence importante sur les instructions de saut conditionnel (également appelées instructions de branchement). Ces instructions permettent, ou non, de continuer l'exécution à un autre endroit du programme, suivant la véracité d'une condition donnée. Le flot de contrôle du programme se scinde donc en deux branches. Dans le cas où la condition est vraie, on dit que le branchement est pris et le programme saute à une adresse donnée ; et dans le cas où la condition est fausse, le programme continue simplement son exécution en exécutant l'instruction qui suit.

Dans le cas d'un processeur pipeliné, ces instructions posent un problème similaire à celui des dépendances entre instructions consécutives, mais sur le flot de contrôle plutôt que sur le flot de données. La question qui est posée est la suivante : comment savoir quelle instruction exécuter après un saut conditionnel sans connaître à l'avance la validité de la condition ?

Comme précédemment, plusieurs options sont envisageables pour résoudre ce problème. La plus simple est d'attendre que la condition soit évaluée pour reprendre l'exécution. C'est une solution qui s'accorderait bien avec une vision logicielle. Cependant, chaque saut conditionnel impliquerait donc à chaque fois plusieurs cycles d'horloge de pénalité (pénalité qui dépend en partie de la profondeur du pipeline). Or, les sauts conditionnels constituent une grande part des programmes informatiques. Ils forment généralement 10 à 20% des instructions d'un programme ; c'est le cas par exemple de la suite de benchmark SPECint2006, composée de 17% d'instructions de branchement [55].

Cette solution est donc la plupart du temps inacceptable. Comme pour les dépendances entre instructions, il faut donc se tourner vers une seconde option. Cette seconde option consiste à prédire la validité de la condition. Ce procédé se nomme *prédiction de branchement*. La prédiction permet de décider en avance de phase si le branchement est pris ou non. Si la prédiction s'avère au final incorrecte, les instructions qui ont commencé à être exécutées sont annulées (leur propagation dans le pipeline est stoppée) et l'exécution reprend au bon endroit. Dans ce cas, il y a la même pénalité que dans le cas où il n'y a aucune prédiction. Si, au contraire, la prédiction s'avère exacte, aucun cycle d'horloge ne sera perdu pendant l'exécution. Tout ce procédé s'appelle *exécution spéculative* : on spécule sur quelles instructions doivent être exécutées. Sur la terminologie utilisée dans cette thèse, nous parlerons d'*instructions correctes* pour qualifier les instructions qui doivent réellement être exécutées par le processeur et d'*instructions fantômes* pour les instructions incorrectement spéculées qui sont annulées avant la fin de leur exécution.

Il existe de nombreuses politiques de prédiction plus ou moins précises, qui exploitent notamment la structure du programme ou l'historique des décisions précédentes. Les meilleurs prédicteurs actuels peuvent atteindre des taux de prédictions correctes supérieurs à 95%, ce qui les rend indispensables dans la majorité des processeurs.
Si l'on reporte le concept d'exécution spéculative au niveau logiciel, il faut considérer qu'à chaque embranchement du programme, les instructions de la mauvaise branche peuvent commencer à être exécutées. Cette caractéristique n'a pas d'effet visible (les instructions incorrectes sont annulées avant leur fin), mais a des effets de bord : elle modifie l'état interne du processeur. Cette modification, nous le verrons dans les sections suivantes, a son importance lorsqu'on considère des injections de fautes.
Cette particularité a par ailleurs été exploitée récemment dans la fameuse attaque SPECTRE [36]. Cette attaque n'est pas une attaque en faute, mais utilise justement la modification de l'état du processeur suite à des instructions incorrectement spéculées. En l'occurrence, la structure observée est la mémoire cache.

III.2.5. Conclusion

Toutes les optimisations évoquées dans cette partie font que l'exécution d'une instruction au niveau matériel est très différente de la vision que nous en avons au niveau logiciel, où les instructions sont considérées atomiques. De cette non-atomicité des instructions au niveau matériel résultent des difficultés résolues au moyen d'un accroissement de la complexité du processeur ; complexité qui rend l'analyse au niveau logiciel bien plus ardue.

III.3. Une architecture open-source : RISC-V

Pour illustrer les propos théoriques exposés jusqu'ici, nous nous tournons maintenant vers une étude pratique, avec un processeur d'architecture RISC-V. En effet, ne compte pas seulement la présence de ces optimisations, mais également leur implémentation concrète. Une vision purement théorique des choses pourrait amener des imprécisions et des conclusions déconnectées de la réalité matérielle ; ce que nous voulons justement éviter.

Nous commencerons par présenter l'architecture RISC-V, puis nous nous attarderons sur les implémentations du pipeline et de la structure de forwarding qui permettront par la suite de comprendre comment les fautes peuvent impacter le processeur.

III.3.1. Présentation de l'architecture RISC-V

RISC-V est une architecture de processeur de type RISC (Reduced Instruction Set Computer) ouverte et libre. Elle est développée depuis 2010 par l'université de Californie à Berkeley. Cette architecture se veut un nouveau standard pour les processeurs RISC.

L'architecture RISC-V est modulaire : elle est composée principalement d'un jeu d'instruction pour les données entières (I), à laquelle plusieurs extensions peuvent être attachées. Parmi celles-ci, il existe notamment une extension pour les multiplications et divisions entières (M), une pour les opérations mémoire atomiques (A), une pour les données à virgule flottante à précision simple (F) ou double (D). Le jeu d'instructions de base avec ces quatre extensions (IMAFD) est plus communément appelé processeur d'usage général (G).

Cette architecture utilise une architecture load/store, ce qui signifie que les instructions opérant sur les registres et sur la mémoire sont découplées. L'architecture est conçue pour des tailles de données de 32, 64 ou 128 bits. Les instructions ont une taille fixe de 32 bits.

L'architecture est constituée d'un jeu de 32 registres. Le registre zéro est fixé à la valeur constante 0.

III.3.2. Implémentation LowRISC

L'implémentation « canonique » de l'architecture RISC-V, développée par la même université, est appelée Rocket. Elle est composée d'un pipeline de cinq étages qui implémente la variante G pour des données de 64 bits. Les rôles principaux de ces cinq étages sont :

- *Instruction Fetch* (IF) : l'instruction à exécuter est cherchée en mémoire.
- *Instruction Decode* (ID) : l'instruction est décodée, les signaux de contrôle sont assignés et des valeurs sont lues dans le banc de registres si besoin.
- *Execute* (EX) : les calculs nécessaires à l'instruction sont effectués dans l'ALU.
- *Memory* (MEM) : les opérations de lecture/écriture vers la mémoire sont effectuées.
- *Write-Back* (WB) : le résultat de l'instruction est écrit dans le banc de registre, si nécessaire.

Pendant cette thèse, c'est l'implémentation LowRISC qui a été utilisée ; cette implémentation est basée sur le cœur Rocket.

Afin de lever les ambiguïtés concernant certaines appellations, nous appellerons registres assembleur les registres contenus dans le banc de registres. Ce sont les registres qui sont manipulés par le code assembleur. Nous appellerons simplement « registres » les autres bascules de la microarchitecture, qui comprennent par exemple les signaux de contrôle des instructions ou tout autre signal interne du processeur. Ces registres sont appelés aussi registres cachés car ils ne sont pas visibles d'un point de vue logiciel, ils sont spécifiques à la microarchitecture.

III.3.2.1. Pipeline

L'implémentation LowRISC est malheureusement peu documentée. En observant le code source de l'implémentation, nous pouvons cependant dessiner une partie de la microarchitecture du processeur. Nous présentons dans la Figure 10 une partie de la microarchitecture de la version 0.2 de l'implémentation, publiée en décembre 2015. Sur cette figure sont détaillées les structures principales des étages EX, MEM et WB, qui seront réutilisées dans la suite de cet exposé. Les étages IF et ID ne sont pas détaillés car moins intéressants pour notre étude ; des attaques dans ces étages provoqueraient majoritairement des sauts dans le programme et des remplacements d'instructions, c'est-à-dire des effets qu'il est possible d'étudier sans forcément avoir une bonne connaissance de l'implémentation matérielle.

Sur la figure, les rectangles ID/EX, EX/MEM et MEM/WB représentent les registres utilisés pour stocker les différents signaux de l'architecture entre les étages du pipeline. Certains noms de signaux sont utilisés plusieurs fois, dans différents étages (c'est le cas par exemple du signal *Reg_write*). Dans ce cas, ces signaux sont bien différentes instances, mais qui remplissent le même rôle dans les différents étages ; ces signaux sont juste transmis d'un étage à l'autre en même temps que l'instruction à laquelle ils sont associés.

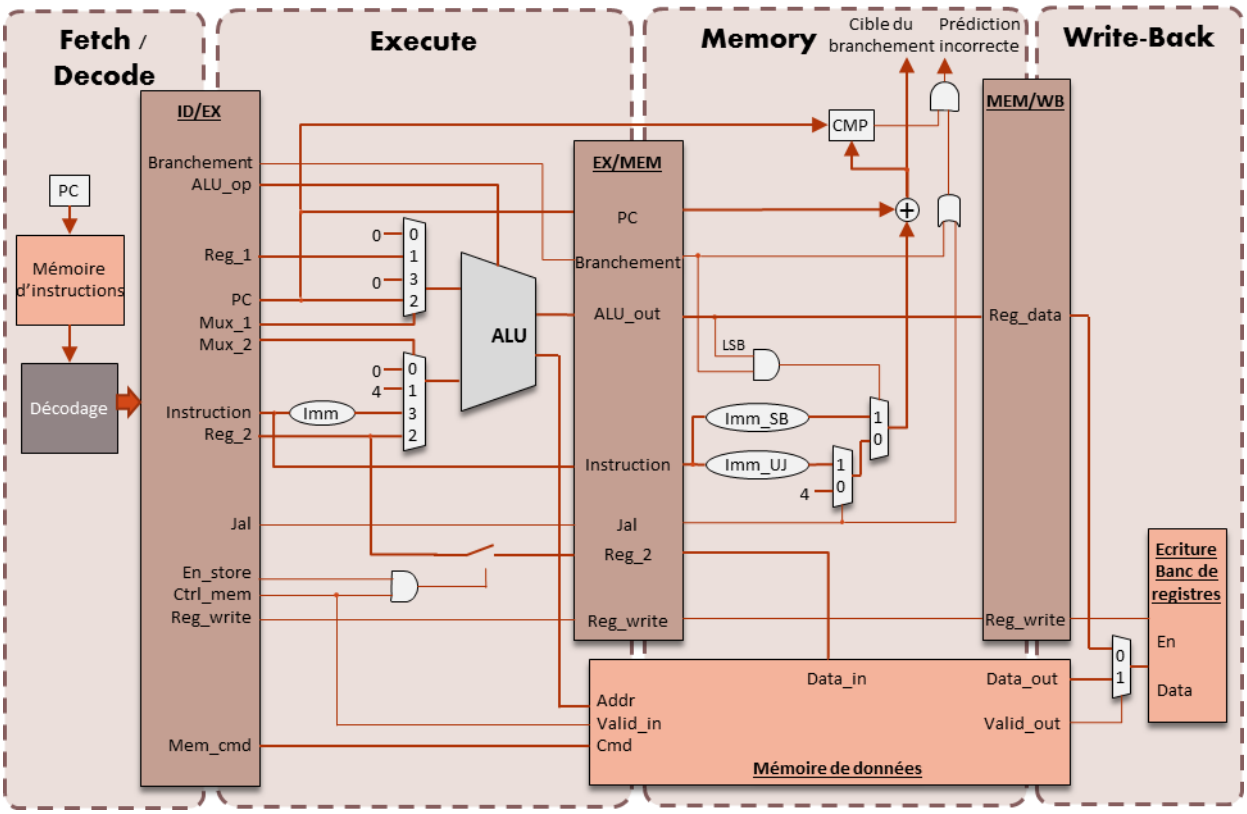

FIGURE 10 : MICROARCHITECTURE DU PROCESSEUR LOWRISC 0.2

Exemple d'exécution d'une instruction

Afin de mieux comprendre comment ce pipeline fonctionne, et donc, plus tard, quels sont les comportements fautifs qui pourront être générés, nous proposons ici d'analyser l'exécution d'une instruction BEQ, qui signifie « Branch if EQual ». Cette instruction compare le contenu de deux registres assembleur, et s'ils sont égaux, le programme saute à un endroit donné du programme. La syntaxe de cette instruction est la suivante : « `beq a1, a2, offset` », avec *a1* et *a2* les deux registres assembleur à comparer et *offset* la longueur du saut à effectuer (relatif à l'adresse actuelle de l'instruction).

Après décodage de l'instruction, les registres ID/EX suivants sont utilisés :

- « Branchement » : permet de savoir que nous avons affaire à une instruction de branchement
- « ALU_op » : permet de spécifier quelle opération doit effectuer l'ALU
- « Reg_1 » : contient la valeur du premier registre assembleur (ici, a1)
- « PC » : contient l'adresse de l'instruction actuelle
- « Mux_1 » : permet de sélectionner d'où vient le premier argument de l'ALU
- « Mux_2 » : permet de sélectionner d'où vient le second argument de l'ALU
- « Instruction » : contient le mot d'instruction de 32 bits
- « Reg_2 » : contient la valeur du deuxième registre assembleur (ici, a2)

Dans l'étage Execute, les valeurs de « Reg_1 » et « Reg_2 » sont passées à l'ALU par l'intermédiaire de deux multiplexeurs. Dans notre cas, les signaux de sélection « Mux_1 » et « Mux_2 » ont les valeurs 1 et 2, respectivement. Le signal « ALU_op » permet de demander à l'ALU de comparer les deux valeurs qui lui sont données en entrée. Il enverra sur sa sortie « Out » la valeur 1 si les valeurs

sont égales ; 0 sinon. Ce résultat est enregistré dans le registre EX/MEM nommé « ALU_out ». En parallèle, les signaux « Branchement », « PC » et « Instruction » sont simplement passés tels quels aux registres EX/MEM du même nom.

C'est dans l'étage Memory que se passe la partie la plus complexe de l'exécution de l'instruction BEQ. La comparaison a eu lieu dans l'étage précédent et il faut maintenant décider s'il faut ou non prendre le branchement. Tout d'abord, l'adresse cible du branchement est calculée. Cette adresse est la somme de l'adresse courante (contenue dans le registre PC) et d'un offset. Cet offset est sélectionné par une série de multiplexeurs. Par défaut, l'offset est 4, ce qui correspond à l'adresse de l'instruction suivante (car chaque instruction a une taille de 4 octets). Quand le résultat de l'ALU est 1 (dans notre cas, quand les deux registres assembleur sont égaux), le second multiplexeur choisit la voie du haut, dont la valeur est obtenue directement à partir du mot d'instruction. Il existe différents formats d'offset en fonction du type d'instruction considéré. Le format Imm_SB est utilisé pour les instructions de branchement. Ce format est représenté en Figure 11. Le rôle du bloc Imm_SB dans le schéma de l'architecture est de collecter les bits « imm » pour reconstituer la valeur de l'offset.

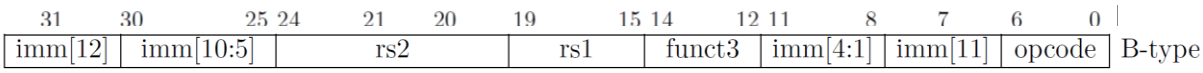

FIGURE 11 : FORMAT DES INSTRUCTIONS DE BRANCHEMENT POUR UNE ARCHITECTURE RISC-V. L'OFFSET DE L'ADRESSE DE BRANCHEMENT EST OBTENU EN CONCATENANT LES CHAMPS MARQUES « IMM » DANS LE MOT D'INSTRUCTION

Comme le processeur utilise l'exécution spéculative, pendant toutes ces opérations, d'autres instructions auront commencé à être exécutées dans les étages précédents. Le but est donc de vérifier si la prédiction était correcte. Pour cela, le résultat du calcul d'adresse expliqué précédemment est comparé à l'adresse de l'instruction actuellement en cours d'exécution dans l'étage Execute. Si la prédiction était correcte, l'exécution peut continuer normalement. Si la prédiction était incorrecte, les étages précédents du pipeline sont invalidés et l'exécution reprend à l'adresse calculée. Dans les deux cas, l'instruction BEQ est ensuite terminée ; il n'y a aucune action effectuée dans l'étage Write-Back car il n'y a pas de valeur à écrire dans le banc de registres.

D'autres types d'instructions

Il existe évidemment d'autres types d'instructions. Pour ne pas alourdir cet exposé, nous ne détaillerons pas leur fonctionnement et nous nous contenterons de quelques remarques générales pour le lecteur intéressé.

Le premier type d'instructions auquel nous pouvons penser concerne les opérations entre registres assembleur. Ces instructions sont appelées instructions de type R. Le fonctionnement dans l'étage EX est similaire aux instructions de branchement. Aucune opération n'est effectuée dans l'étage MEM, puis les résultats sont écrits dans le banc de registres dans l'étage WB.

Des opérations peuvent également avoir lieu entre un registre assembleur et une valeur immédiate. Ces instructions sont de types I. Leur fonctionnement est presque identique aux instructions de type R, à la différence que le multiplexeur du deuxième argument de l'ALU sélectionne un chemin différent. La valeur immédiate est directement obtenue à partir de certains champs du mot d'instruction.

Pour les opérations de lecture et d'écriture dans la mémoire, l'adresse est calculée dans l'étage EX, en additionnant la valeur d'un registre assembleur avec une valeur immédiate. Cette valeur ainsi que les autres signaux de contrôle sont également envoyés à la mémoire dans cet étage. Pour les instructions d'écriture, la valeur à écrire est envoyée dans l'étage MEM. Pour les opérations de lecture, la valeur est retournée dans l'étage WB et directement écrite dans le banc de registres.

En plus des sauts conditionnels, il existe aussi des sauts inconditionnels. Dans l'architecture RISC-V, ces sauts réalisent deux fonctions en même temps : sauter à un autre endroit du programme et sauvegarder en même temps l'adresse suivant l'instruction dans un registre assembleur. Cette deuxième fonction est utile dans le cas où le saut est un appel à une fonction.

Enfin, il existe aussi des instructions sur les registres de statut et de contrôle (CSR). Ces registres ont en charge notamment les différents niveaux de privilèges (machine, hyperviseur, superviseur, utilisateur). Ces instructions n'ont cependant pas été étudiées car elles touchent à des éléments en lien avec le système d'exploitation, plus qu'avec les programmes eux-mêmes. Le travail était plutôt focalisé sur des structures plus standards des processeurs.

III.3.2.2. Forwarding

Une des structures montrant le plus de comportements fautifs intéressants est celle qui concerne l'optimisation de forwarding. Il est donc intéressant de comprendre également son implémentation. Celle-ci est présentée en Figure 12.

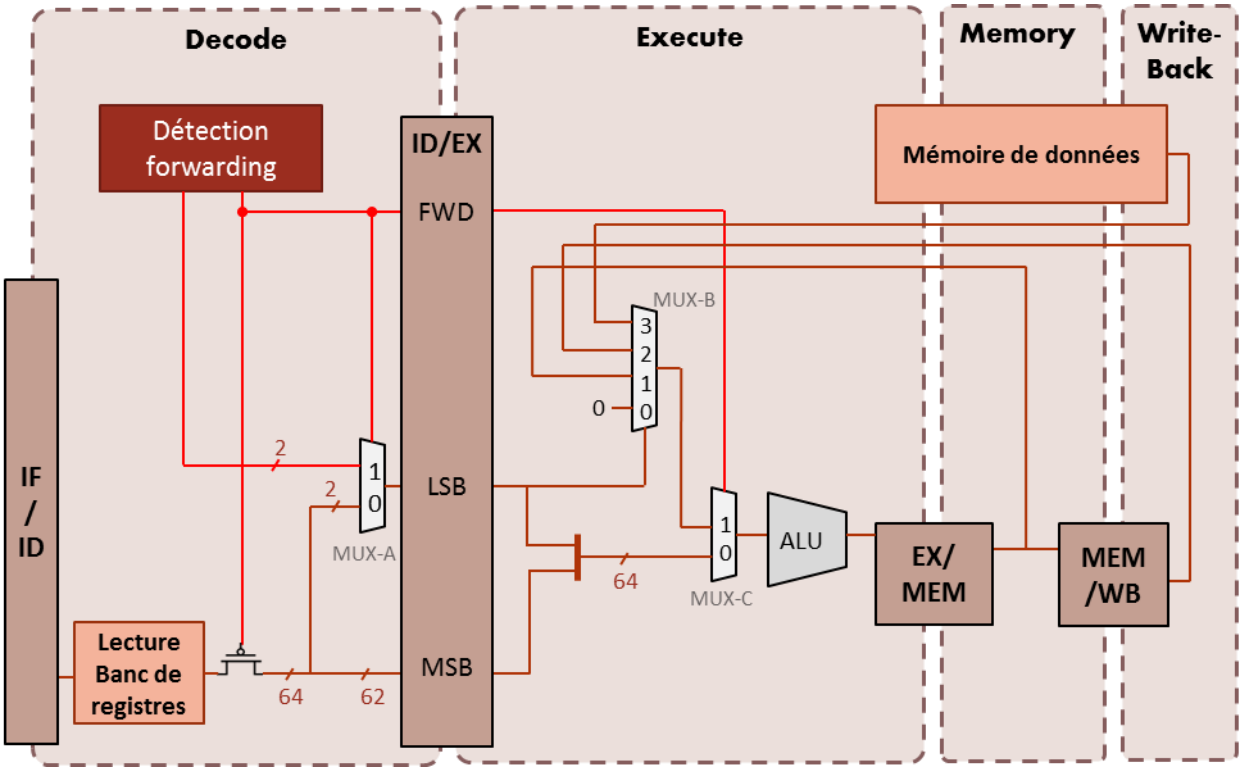

FIGURE 12 : MICROARCHITECTURE DE LA STRUCTURE DE FORWARDING, POUR UN ARGUMENT

Sur cette figure nous pouvons tout d'abord reconnaître l'ALU. C'est toujours la même que dans le schéma global de la microarchitecture, mais par manque de place, la structure de forwarding est présentée indépendamment. La Figure 12 représente le forwarding pour un argument seulement. Comme le forwarding peut apparaître sur les deux arguments d'une instruction, la structure de la Figure 12 est en réalité présente deux fois dans le processeur. Cette structure s'insère juste en amont

des multiplexeurs devant l'ALU dans la Figure 10, à la place des signaux « Reg_1 » et « Reg_2 ». Attention cependant, la Figure 12 détaille également le contenu de l'étage de décodage car le fowarding agit dès cet étage.

Dans la section précédente, nous avons présenté le fonctionnement de l'instruction BEQ, en considérant que les valeurs des registres étaient dans les signaux « Reg_1 » et « Reg_2 ». Nous allons voir maintenant comment ces valeurs sont assignées.

Tout d'abord, dans le cas où il n'y a pas de situation de forwarding (pas de dépendance de l'instruction courante aux instructions précédentes), une valeur est lue dans le banc de registres puis est séparée en deux signaux MSB et LSB qui contiennent respectivement les 62 bits de poids fort et les 2 bits de poids faible (pour rappel les données ont une taille de 64 bits). Ces signaux MSB et LSB sont ceux qui sont enregistrés dans les registres ID/EX. Dans l'étage Execute, ces deux parties sont recombinées puis envoyées dans l'ALU.

Dans le cas d'une situation de forwarding, la situation est plus complexe. Le forwarding est détecté par une structure visible sur le schéma, mais non détaillée. Cette détection est basée sur une comparaison du numéro de registre assembleur source avec le numéro des registres assembleur destination des étages suivants. Si la destination d'une des instructions dans les étages Execute ou Memory est la même qu'une des sources de l'instruction dans l'étage de décodage, alors il y a forwarding. Dans ce cas, trois opérations sont réalisées :

- La lecture dans le banc de registre est bloquée, ce qui a pour conséquence directe que le registre MSB n'est pas mis à jour ; il conserve sa valeur actuelle.
- La détection du forwarding génère un signal de 2 bits qui permet de sélectionner, grâce au multiplexeur B, la valeur qui doit être forwardée. Si ce signal vaut 3, la valeur doit être prise à la sortie de la mémoire (la dépendance est sur une instruction de lecture mémoire). S'il vaut 2, la valeur est le résultat de l'instruction qui sera dans l'étage WB. S'il vaut 1, la valeur est le résultat de l'instruction qui sera dans l'étage MEM. Enfin, s'il vaut 0, c'est que le registre demandé était le registre 0 (ce qui ne correspond en réalité pas à une situation de forwarding, mais est une particularité de l'implémentation). Le signal de 2 bits généré est stocké dans le registre LSB par l'intermédiaire d'un multiplexeur.
- Le registre FWD prend la valeur 1 pour indiquer qu'il y a une situation de forwarding.

Dans l'étage Execute, le signal LSB est utilisé pour sélectionner avec le multiplexeur B la valeur qui doit être utilisée et le signal FWD permet de faire passer cette valeur à l'ALU en lieu et place de la recombinaison des signaux LSB et MSB, grâce au multiplexeur C.

A propos de cette structure, un peu complexe, nous pouvons faire plusieurs remarques. D'une part, le signal MSB n'est utilisé que dans les situations où il n'y a pas de forwarding ; il n'est pas mis à jour quand une situation de forwarding est détectée. D'autre part, le signal LSB joue deux rôles suivant la situation : il contient soit les bits de poids faible d'un des arguments de l'opération, soit le signal de sélection d'un multiplexeur. Ces deux points auront une conséquence sur l'impact des fautes.

III.4. Comportements fautifs observés

Maintenant que l'implémentation du processeur a été étudiée, il est possible de comprendre en détail comment le processeur réagit à des injections de fautes. Pour cela, nous proposons de réaliser diverses injections dans le processeur.

Nous détaillerons tout d'abord comment ces injections ont été réalisées. Puis nous nous intéresserons aux comportements provoqués par les fautes dans le processeur et leur impact sur des contremesures classiques et sur une application sécurisée.

III.4.1. Paramètres d'injection

Pour injecter des fautes au niveau RTL, il est nécessaire de définir tout d'abord le modèle de faute RTL utilisé, ainsi que la méthode d'injection.

Dans cette thèse, c'est le modèle du bit-flip qui est sélectionné. Le choix de ce modèle plutôt que des modèles de bit-set et bit-reset s'explique de la façon suivante. Les modèles de bit-set et bit-reset sont particulièrement utiles dans une optique d'analyse de faute différentielle, ou lorsqu'il y a un apriori sur les moyens d'injection de l'attaquant. Nous nous intéressons au contraire à une modélisation plus générale des effets qu'il est possible d'obtenir dans un processeur. Notre but est de provoquer le plus de situations possibles afin de mieux comprendre les comportements fautifs de ce processeur. C'est en cela que le modèle du bit-flip est plus intéressant, puisqu'il permet toujours d'avoir un effet sur le circuit, là où, par exemple, un bit-reset n'a aucun effet si la bascule visée est déjà à zéro. De ce point de vue de modélisation, le bit-flip englobe tous les effets provoqués à la fois par les bit-sets et les bit-resets.

Les fautes injectées sont de multiplicité un, ce qui s'accorde plutôt avec un moyen d'injection précis, tel que le laser. Des injections multiples aléatoires seront également considérées plus tard dans ce manuscrit.

Les injections sont réalisées en simulation, dans le logiciel QuestaSim de Mentor Graphics. Comme le modèle du bit-flip est très simple et peut tout à fait être injecté avec des commandes simulateur, c'est cette solution qui a été retenue. Elle permet de s'affranchir de la difficulté à modifier la description matérielle du processeur pour insérer des mutants ou des saboteurs.

En plus de ces paramètres, il faut également définir la cible des injections, tant au niveau matériel (dans quelle bascule injecter ?) qu'au niveau logiciel (pendant l'exécution de quel programme ?).

Comme le but de cette section est simplement de montrer quels types de comportements fautifs il est possible d'obtenir, les lieux d'injection n'étaient pas aléatoires, mais précisément sélectionnés pour présenter des comportements fautifs intéressants. Ainsi, les bascules étaient choisies parmi les bascules des étages Execute, Memory ou Write-Back. De même, les programmes exécutés par le processeur étaient, dans un premier temps, des petits segments de code assembleur spécialement conçus pour mettre le processeur dans des états spécifiques. Ces programmes ne sont pas détaillés dans ce chapitre, mais le seront dans le chapitre suivant où les injections sont automatisées.

III.4.2. Quelques comportements fautifs

Dans cette section, quelques premiers comportements fautifs obtenus en simulation sont présentés. Les fautes sont présentées en trois parties : d'abord les fautes sur le pipeline présenté dans la Figure 10, puis les fautes qui exploitent l'optimisation de forwarding et enfin des fautes en lien avec l'exécution spéculative.

III.4.2.1. Fautes dans le pipeline

Comme le comportement de l'instruction BEQ a été explicité en détail grâce à la Figure 10, nous pouvons commencer par nous intéresser à l'injection de fautes lors de l'exécution de cette instruction. Une première faute simple peut consister à fauter le signal « branch » pour le mettre à

zéro. Dans ce cas, les multiplexeurs dans l'étage MEM sélectionneront toujours la valeur 4 pour le calcul de l'adresse d'instruction, ce qui permet de forcer l'exécution de l'instruction qui suit, même si le branchement devait être pris. D'un point de vue haut niveau, cela correspond tout simplement à ne pas exécuter le saut conditionnel. Nous voyons ici une première manière d'effectuer un saut d'instruction (restreint aux instructions de branchement).

Un deuxième exemple de faute est d'impacter les signaux « Mux_1 » ou « Mux_2 ». Dans ce cas, la valeur envoyée à l'ALU n'est plus la valeur du registre assembleur. Nous pouvons voir sur la Figure 10 que dans tous les cas, il est possible d'envoyer la valeur 0. A la place de comparer les registres assembleur a1 et a2, il est donc possible d'injecter une faute pour comparer a1 à 0 ou a2 à 0. Ce comportement avalise donc l'utilisation de la valeur 0 fréquemment utilisée dans les modèles de faute logiciels.

D'autres comportements ont pu être trouvés. Nous ne les détaillons pas tous ici pour ne pas trop alourdir les propos, mais le Tableau 2 présente une liste de comportements fautifs, en fonction du type d'instruction et de la bascule visée. Les noms des bascules correspondent aux noms introduits dans la Figure 10. Le lecteur intéressé pourra ainsi comprendre plus en détail comment ces comportements sont créés en se référant à cette figure. Cette liste est non exhaustive ; l'objectif de ce chapitre étant seulement d'avoir une idée du genre de comportements qu'il est possible d'obtenir. La modélisation plus complète des comportements obtenus est l'objet du chapitre suivant.

Parmi les comportements fautifs observés, certains ont des effets assez proches de ceux prédits par des modèles de faute logiciels classiques ; mais d'autres ont des effets particuliers qu'il est intéressant de comprendre plus précisément. Ils sont numérotés dans le tableau et leurs causes ou conséquences sont discutées dans les sections suivantes.

TABLEAU 2: LISTE DE COMPORTEMENTS FAUTIFS OBTENUS, EN FONCTION DU TYPE D'INSTRUCTION ET DU LIEU DE L'INJECTION. LES COMPORTEMENTS NUMEROTES SONT DISCUTES PAR LA SUITE, DANS LES SECTIONS CONCERNANT 1) LES COMPORTEMENTS HYBRIDES ; 2) LES OPERATIONS DE L'ALU ; 3) LES FAUTES INDIRECTES DE FORWARDING ; ET 4) LA REACTIVATION D'INSTRUCTIONS INCORRECTEMENT SPECULEES.

Instruction	Lieu d'injection	Comportement fautif
Branchement	Branchement	Empêcher le branchement d'être pris
	Mux_1 ou Mux_2	Comparaison à 0 au lieu d'un des arguments
	ALU_op	Inversion de la condition de test(2)
	Reg_write	Opération normale, mais un registre assembleur prend aussi la valeur 0 ou 1(1)
	(non representé)	Exécution de l'instruction suivante, même si le branchement est pris(4)
Type R	Reg_write	Empêcher le résultat d'être écrit dans le banc de registres(3)
	Branchement	Saut en plus de l'opération normale (seulement si le résultat de l'ALU est impair)(1)
	Mux_1 ou mux_2	Remplacer un argument par 0
	ALU_op	Effectuer une autre opération mathématique(2)
Lecture mémoire	Reg_write	Empêcher la valeur lue d'être écrite dans le banc de registres(3)
	Ctrl_mem	Empêcher la lecture et écrire l'adresse dans le registre assembleur destination.
	ALU_op	Soustraction au lieu d'une addition pour le calcul de l'adresse.
	Mem_cmd	Ecriture de la dernière donnée écrite en mémoire et écriture de l'adresse dans le registre assembleur destination.
	Mem_cmd	Opération de lecture normale, puis écriture de la dernière donnée écrite en mémoire
	Mem_cmd	Opération de lecture normale, puis écriture en mémoire de la somme de la valeur lue et de la dernière donnée écrite.
Ecriture mémoire	Ctrl_mem	Empêcher l'écriture.
	ALU_op	Soustraction au lieu d'une addition pour le calcul de l'adresse.
	Reg_write	Opération d'écriture normale et écriture de l'adresse dans un registre assembleur (qui dépend de l'offset de l'adresse).
	En_store	Ecriture de la dernière donnée écrite plutôt que celle demandée.
	Mem_cmd	Ecriture de la nouvelle donnée XOR la dernière donnée écrite
Saut inconditionnel (jal)	Reg_write	Empêcher l'écriture de l'adresse de retour dans le banc de registre.
	Mux_2	Ecriture de PC au lieu de PC+4 pour l'adresse de retour.
	Jal	Empêcher le saut
	(non representé)	Exécution de l'instruction suivant directement le saut(4)

*Comportements hybrides et registres/valeurs sensibles (comportements notés **(1)** dans le tableau)*

Commençons par discuter les comportements notés **(1)** dans le tableau précédent. Une faute sur le signal *Reg_write* pendant l'exécution d'une instruction de branchement a pour effet d'écrire une valeur dans le banc de registre. Deux choses doivent être précisées : d'une part quelle valeur est écrite et d'autre part dans quel registre assembleur cette donnée est écrite. En observant le schéma de la microarchitecture, nous nous rendons compte que la valeur écrite est le résultat de l'ALU ; ce sera donc le résultat de la comparaison. En ce qui concerne le registre dans lequel le résultat est écrit, il faut à nouveau se tourner vers le format des instructions de branchement (Figure 11). Dans le format d'instruction RISC-V, le registre assembleur de destination est toujours placé dans les bits 7 à 11, ce qui correspond pour une instruction de branchement à une partie de l'offset. En observant plus attentivement le format de l'instruction, nous pouvons en déduire que ce sont les registres

multiples de 4 qui seront les plus facilement touchés par cette attaque car ils correspondent à des offsets relativement courts. D'autres registres peuvent être touchés, mais moins vraisemblablement car ils correspondent à des offsets très grands. Enfin, certains registres ne peuvent être touchés par cette attaque. Pour résumer, lorsqu'une instruction de branchement est exécutée, une faute permet de forcer un registre assembleur multiple de 4 à la valeur 0 ou 1, suivant si la condition est fausse ou vraie (respectivement). Cette conclusion est intéressante dans la mesure où elle indique que certains registres assembleur peuvent être plus sensibles que d'autres et que certaines valeurs (0 ou 1) peuvent être plus faciles à obtenir que d'autres.

Le comportement précédent consiste donc à ajouter à une instruction de branchement un peu de la fonctionnalité d'une instruction de type R (écriture dans un registre). Il est également possible de faire l'inverse : partir d'une instruction de type R et lui ajouter de la fonctionnalité d'une instruction de branchement. C'est le rôle de la faute sur le signal *branchement*. En plus de réaliser l'opération d'écriture dans un registre, le programme saute à un autre endroit du programme. Tout comme dans le cas précédent, il faut s'intéresser au format de l'instruction pour déduire vers quel endroit le saut est effectué : le mot d'instruction est à nouveau interprété, sous une autre forme. Ainsi, pour avoir une cible valide pour le saut, il faut que l'instruction visée (de type R) écrive dans un registre assembleur multiple de 4 ou égal à 1 modulo 4. A noter également que pour que le saut s'effectue, il est nécessaire que le résultat de l'ALU soit impair (car tout comme dans une instruction de branchement, c'est le bit de poids faible du résultat de l'ALU qui est utilisé pour dire si le branchement est pris ou non).

Ces deux comportements font donc apparaître des instructions hybrides. Par l'injection de faute, le processeur est mis dans un état qui ne correspond pas à l'exécution d'une instruction valide, mais dans un état intermédiaire, en combinant des fonctionnalités de deux types d'instruction différents.

*Opérations de l'ALU (comportements notés **(2)** dans le tableau)*

Les comportements discutés dans cette section sont en rapport avec les fautes notées **(2)** dans le Tableau 2.

L'opération effectuée par l'ALU est choisie grâce à un signal de 4 bits appelé *ALU_op*, comme représenté sur la Figure 10. En impactant ce signal avec une faute, il est possible de modifier l'opération effectuée. Pour se rendre compte de quelles transformations peuvent être obtenues dans le contexte de fautes de multiplicité 1, nous pouvons nous référer au Tableau 3, qui expose les différentes opérations possibles en fonction du signal *ALU_op*. Le tableau est arrangé suivant le code Gray afin de mieux visualiser les transformations de multiplicité 1 : pour chaque opération, les transformations possibles sont les quatre cases adjacentes. Ainsi, une addition (add) peut être changée en XOR, en décalage vers la gauche (sl : shift left), en comparaison d'égalité (seq : set if equal) ou même en une opération non reconnue (-). Cette dernière opération ne peut pas arriver dans une exécution normale, mais peut être provoquée par une faute. Pour déduire son effet exact, il est nécessaire d'analyser précisément l'implémentation de l'ALU ; nous ne nous attarderons pas sur ce point ici.

TABLEAU 3 : OPERATIONS DE L'ALU EN FONCTION DU SIGNAL *ALU_OP*

Bits de poids faible → / Bits de poids fort	00	01	11	10
00	add	sl	-	-
01	xor	sr	and	or
11	slt	sge	sgeu	sltu
10	seq	sne	sra	sub

Il y a deux choses intéressantes à remarquer dans ce tableau. Chaque opérateur de comparaison est adjacent à son complément (seq/sne, sge/slt et sgeu/sltu). C'est une manière efficace de concevoir l'ALU, mais cela signifie aussi qu'un simple bit-flip peut inverser la condition d'un test (ce qui au passage confirme la faisabilité du modèle logiciel d'inversion de test). La seconde remarque intéressante est que chaque opérateur arithmétique est adjacent à un opérateur de comparaison. Or les comparaisons ont toujours pour résultat 0 ou 1. Cela montre une nouvelle fois que ces deux valeurs sont plus faciles à forcer dans un programme qu'une valeur quelconque et que par conséquent elles devraient être maniées avec précaution. En particulier, cette remarque donne du poids à l'utilisation de booléens durcis pour les opérations booléennes, comme évoqué en section II.3.3.3.

Pour conserver le même plan que dans la partie sur l'analyse théorique de l'architecture, les comportements notés **(3)** et **(4)** dans le Tableau 2 sont discutés plus loin dans les sections suivantes car ils correspondent à des comportements en lien avec le pipeline mais également avec le *forwarding* et l'exécution spéculative.

III.4.2.2. Fautes mettant en jeu le forwarding

Fautes directes sur la structure de forwarding

Le Tableau 2 a permis d'introduire différents comportements qu'il est possible de provoquer dans le pipeline du processeur. Comme évoqué dans la section III.2, un processeur possède également plusieurs optimisations, au premier rang desquelles on trouve le forwarding. Cette structure d'optimisation fait là aussi apparaître des comportements nouveaux.

Pour le processeur étudié, nous avons pu voir en Figure 12 que la structure de forwarding met en jeu les signaux suivants : *MSB*, *LSB* et *FWD*.

Fauter le signal *MSB* n'a que peu d'intérêt : dans le cas où il n'y a pas de situation de forwarding, l'effet de la faute serait simplement une modification de la valeur d'un argument (ce qui est certes un problème, mais qui peut être aisément anticipé au niveau logiciel). Dans le cas où il y a une situation de forwarding, le signal n'est tout simplement pas utilisé.

Fauter le signal *LSB* est plus intéressant. Dans le cas où il n'y a pas de situation de forwarding, l'effet est une simple modification de l'argument, mais dans le cas où il y a une situation de forwarding, l'effet d'une faute est alors de changer l'étage qui doit être forwardé. Nous pouvons notamment transmettre la valeur 0, ou la dernière valeur lue en mémoire. En effet, dans l'implémentation du processeur utilisé, la dernière valeur lue reste disponible en sortie de la mémoire. En temps normal,

cela ne pose pas problème car cette valeur n'est jamais lue, mais quand l'injection de faute est prise en compte, cette caractéristique peut devenir problématique.

Enfin, une faute sur le signal *FWD* est un peu plus complexe à analyser. Dans le cas où il n'y a pas de situation de forwarding, le forwarding est forcé, en choisissant l'étage en fonction des deux bits de poids faible de l'argument qui vient du banc de registres. Dans le cas où il y a une situation de forwarding, alors le forwarding est désactivé et la valeur qui est transmise à l'ALU est une combinaison des deux bits qui étaient sensé indiquer l'étage à forwarder, avec les 62 bits de poids fort de la dernière valeur qui a transité dans le registre *MSB* (c'est-à-dire la dernière valeur non forwardée sur cet argument). Il y a ici une asymétrie dans les arguments des instructions. En effet seules les instructions de type R et les instructions de branchement ont besoin d'un deuxième registre assembleur source. Par conséquent, le signal *MSB* du deuxième argument de l'ALU est beaucoup moins utilisé que celui du premier argument, ce qui signifie que les valeurs qui sont utilisées dans le deuxième argument peuvent être retenues pendant plus longtemps sans être écrasées. Cette caractéristique peut poser problème si cette valeur est sensible.

*Fautes indirectes de forwarding (comportements notés **(3)** dans le tableau)*

Nous avons parlé jusqu'ici de fautes qui visaient directement des registres mis en jeu dans la structure de forwarding. Mais il est également possible de se servir de cette optimisation pour créer des effets secondaires sur des comportements déjà évoqués dans la section précédente. Ces effets secondaires peuvent être provoqués notamment sur les comportements notés **(3)** dans le Tableau 2.
Prenons la faute *Reg_write* pour les instructions de type R. La faute impacte le signal qui permet de demander l'écriture dans le banc de registres. En injectant un bit-flip pendant une instruction de type R, nous empêchons donc cette écriture. Or le but d'une instruction de type R est justement d'écrire un résultat dans le banc de registres. Nous pourrions alors considérer que cette faute crée un simple saut d'instruction, ce qui est vrai si l'on considère une exécution isolée de l'instruction. Mais en considérant le contexte d'exécution de l'instruction, l'impact réel devient plus complexe.
Dans la Figure 10, nous avons pu remarquer qu'il existe différents signaux *Reg_Write,* dans les étages Execute, Memory et Write-Back. Comme ce signal est simplement transmis d'un étage à l'autre, injecter la faute dans n'importe quel étage aura le même effet sur l'instruction visée. En fonction de l'étage visé, l'impact sur les instructions suivantes pourra cependant être différent. En effet, si les instructions suivantes ont une dépendance au résultat de l'instruction fautée, le résultat pourra quand même être correct car la valeur aura été forwardée avant que l'injection ait lieu.
Considérons le court programme de la Figure 13. Dans ce programme, le registre **a0** est d'abord mis à zéro, puis incrémenté une fois ; puis les registres **a1**, **a2**, **a3** et **a4** prennent la nouvelle valeur de **a0** sommée avec la valeur 1. Le résultat de chaque addition est donné à la fin de chaque ligne. Dans ce programme, exécuté sur le processeur LowRISC, les opérations sur **a1** et **a2** font intervenir le processus de forwarding pour pouvoir utiliser immédiatement la nouvelle valeur de **a0**, sans attendre que la deuxième instruction soit terminée.
Si une faute est injectée dans le registre *Reg_write* pendant l'exécution de la première addition, trois effets différents peuvent se manifester, en fonction de l'étage dans lequel est injectée cette faute. Les instructions ayant un résultat différent de l'exécution saine apparaissent en rouge.

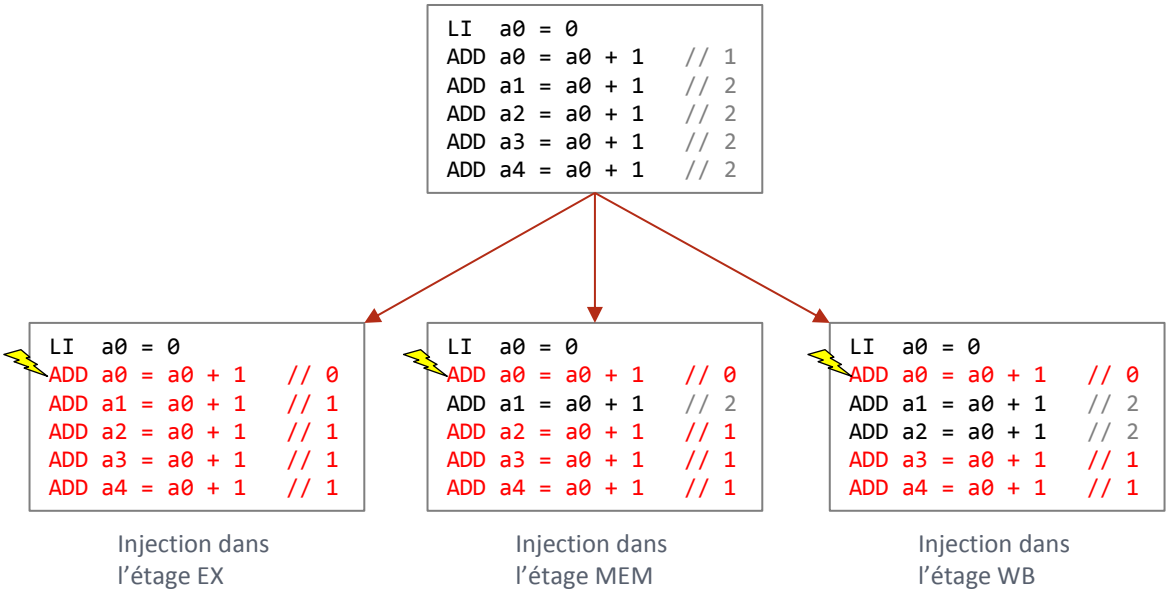

FIGURE 13: CONSEQUENCES D'UNE ATTAQUE SUR LE SIGNAL REG_WRITE, EN FONCTION DE L'ETAGE DE PIPELINE VISE

Le scénario de gauche correspond à un simple saut d'instruction, contrairement aux deux scénarios suivants. Dans ces deux scénarios, certaines instructions suivant l'injection de faute peuvent s'exécuter comme si l'instruction fautée s'était exécutée correctement. La vue du pipeline, représentée en Figure 14 permet de mieux comprendre l'effet de cette faute.

Cycle	IF	ID	EX	MEM	WB
t0	LI a0	-	-	-	-
t1	ADD a0	LI a0	-	-	-
t2	ADD a1	ADD a0	LI a0	-	-
t3	ADD a2	ADD a1	**ADD a0**	LI a0	-
t4	ADD a3	ADD a2	ADD a1	**ADD a0**	LI a0
			Forward		
t5	ADD a4	ADD a3	ADD a2	ADD a1	**ADD a0**
				Forward	
t6	-	ADD a4	ADD a3	ADD a2	ADD a1

FIGURE 14 : VUE DU PIPELINE POUR L'EXECUTION DU PROGRAMME DE LA FIGURE 13.

Nous pouvons voir qu'attaquer pendant que l'instruction est dans l'étage Write-Back (au cycle t5) n'aura pas d'effet sur les deux instructions suivantes car le résultat aura déjà été forwardé. Similairement, attaquer pendant l'étage Memory (au cycle t4) n'aura pas d'effet sur l'instruction **ADD A1** car le résultat aura déjà été forwardé, mais aura un effet sur l'instruction **ADD A2** car l'instruction n'a pas encore été exécutée et l'injection sur le signal *Reg_write* annule en même temps le forwarding. Enfin, injecter pendant l'étage Execute (au cycle t3) aura un effet sur toutes les instructions puisque le résultat n'aura pas encore été forwardé et le forwarding est annulé pour les instructions suivantes.

Ainsi, en se servant du forwarding, il est possible de sauter une instruction sans impacter les instructions qui suivent. Cette caractéristique, nous le verrons, peut rendre caduque certaines contremesures logicielles car elle viole l'hypothèse d'atomicité des instructions faite au niveau logiciel.

III.4.2.3. Fautes mettant en jeu l'exécution spéculative

L'exécution spéculative est une autre optimisation qui crée des comportements intéressants. Lorsque la spéculation se révèle correcte, aucun effet supplémentaire n'est présent ; c'est quand la spéculation est incorrecte que des effets collatéraux prennent place.

*Réactivation d'instructions incorrectement spéculées (comportements notés **(4)** dans le tableau)*

Les effets décrits ici sont en rapport avec les fautes notées **(4)** dans le Tableau 2.

Lors d'une spéculation incorrecte, il est nécessaire d'annuler les instructions qui ont commencé à être exécutées. Dans le processeur utilisé, cette annulation se fait d'une manière relativement simple : les signaux de validation des étages de pipeline concernés sont mis à 0 (ces signaux ne sont pas représentés sur la Figure 10, mais chaque ensemble de registre ID/EX, EX/MEM et MEM/WB possède un signal de validation global qui permet de dire si l'étage est actif ou non). Cette action empêche l'instruction de passer dans les étages suivants du pipeline, mais tous les signaux de contrôle de l'instruction (hormis son signal de validation) conservent leur valeur, jusqu'à ce qu'elles soient écrasées par une instruction future. Pendant plusieurs cycles d'horloge, il est donc possible d'injecter une faute pour réactiver une instruction spéculée et ainsi terminer son exécution. Plus spécifiquement, avec l'implémentation utilisée, il est possible de réactiver la première ou la troisième instruction spéculée.

Pour illustrer cela, nous pouvons considérer le code présenté en Figure 15, en considérant le cas où le branchement est sensé être pris (exécution de Inst_11 après l'instruction de branchement), mais la prédiction est incorrecte (exécution spéculative des instructions Inst_1 et suivantes). Une exécution correcte du code devrait être « BEQ → Inst_11 », mais en injectant une faute bit-flip, il est possible d'obtenir une exécution « BEQ → Inst_1 → Inst_11 » ou une exécution « BEQ → Inst_3 → Inst_11 ».

```
    BEQ a1, a2, label;
    Inst_1
    Inst_2
    Inst_3
    […]
Label:
    Inst_11
    Inst_12
    Inst_13
    […]
```

FIGURE 15 : EXEMPLE DE BRANCHEMENT

Forwarding d'états spéculés

Même si la spéculation d'instructions n'a, en temps normal, pas d'effet visible du point de vue logiciel, elle modifie quand même l'état des registres internes du processeur. Or, dans le processeur, les étages du pipeline ne sont pas entièrement découplés ; en particulier, certaines structures comme le forwarding créent un lien direct entre différents étages. Après une mauvaise spéculation, l'exécution des instructions correctes peut donc être impactée par les états rémanents du passage des instructions fantômes. En utilisant l'attaque de forçage du forwarding, nous pouvons par exemple faire passer le résultat d'une instruction fantôme dans l'instruction correcte. L'instruction fantôme n'est pas exécutée entièrement, mais ses effets et résultats restent présents dans le pipeline.

La durée de rémanence de ces états latents dépend des registres considérés. Certains seront très vite remplacés par les résultats des instructions correctes ; d'autres peuvent rester dans le pipeline pendant plus longtemps, comme le registre MSB du deuxième argument de l'ALU (discuté dans la section précédente).

III.4.2.4. Conclusion sur les comportements fautifs observés

Au travers de ces différents exemples de comportements fautifs, nous avons pu montrer qu'en considérant la microarchitecture du processeur, de nombreux comportements différents peuvent être créés. Certains de ces comportements sortent complètement du cadre des modèles de faute logiciels de l'état de l'art.

Nous avons pu voir en particulier que certaines fautes visent plus facilement certains registres assembleur que d'autres, que certaines valeurs comme 0 ou 1 sont plus faciles à provoquer, qu'il est possible de combiner des fonctionnalités de plusieurs instructions, que certaines fautes ont des effets différents suivant le contexte d'exécution, que certains registres internes peuvent maintenir des valeurs qui peuvent être réutilisées plus tard par injection de faute et qu'il est possible de matérialiser les résultats d'instructions fantômes.

Ces comportements sont spécifiques à l'implémentation du processeur utilisé et ne peuvent pas être anticipés en considérant uniquement le point de vue logiciel.

III.4.3. Conséquences sur des contremesures typiques

Les comportements fautés évoqués jusqu'ici sont intéressants en soit, mais quel est leur impact sur la sécurité d'un système ? Peuvent-ils être utilisés par un attaquant pour casser la sécurité d'une application ?

Il est difficile de répondre à cette question dans l'absolu, car les effets finaux d'une faute dépendent grandement du programme exécuté. Nous nous attacherons cependant à montrer dans cette section quelques contremesures logicielles typiques qui peuvent être contournées au moyen des fautes évoquées précédemment. La section suivante présentera quant à elle des vulnérabilités qui peuvent apparaître dans une application sécurisée de vérification de code PIN.

III.4.3.1. Duplication simple

Dans la partie II.3.3.1, nous avons évoqué une contremesure développée par Moro et al., qui vise à s'assurer qu'un programme puisse correctement s'exécuter même en présence d'un saut d'instruction. Le principe était de remplacer chaque instruction par une séquence équivalente d'instructions idempotentes, puis de dupliquer chacune d'entre elles. L'application de cette contremesure à une instruction XOR est montrée sur la Figure 16.

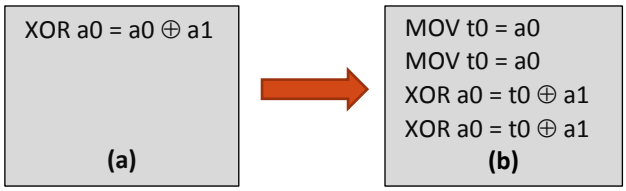

FIGURE 16: (A) INSTRUCTION XOR NON PROTEGEE ; (B) INSTRUCTION XOR PROTEGEE AVEC LA DUPLICATION SIMPLE

Le second XOR doit forwarder la valeur de son premier argument (le registre **t0**) car elle provient de la seconde instruction MOV qui n'est pas encore terminée. Une injection de faute visant à modifier l'étage forwardé peut permettre de transmettre le résultat du premier XOR, comme illustré sur la

Figure 17. Comme appliquer le même argument (ici le registre **a1**) deux fois à un XOR résulte dans la donnée originale, la valeur de a0 à l'issu de la contremesure est toujours la même. Dit autrement, l'injection de faute a permis de retrouver le même comportement qu'un saut d'instruction du XOR non protégé. Bien sûr, cette attaque n'est possible que dans le cas d'un XOR, ce qui limite sa portée. L'exemple montré est intéressant dans le sens où il est possible de créer le même comportement fautif que dans le cas non protégé, par un moyen détourné.

Il faut noter ici une certaine nuance dans le concept de modèle de faute logiciel. Si le saut d'instruction est pris en tant que représentation stricte d'un comportement matériel (comme le fait Moro), alors l'analyse de cette section est non pertinente : la faute que nous injectons n'est pas un saut d'instruction. Si, par contre, le saut d'instruction est considéré comme un comportement problématique du programme (indépendamment du moyen utilisé pour l'obtenir), alors cette analyse a un sens. Si une analyse du programme initial conclut à une vulnérabilité si l'attaquant parvient à effectuer un saut d'instruction, alors peu importe le moyen qu'il utilise pour l'obtenir, c'est bien le comportement final qui crée la vulnérabilité. En d'autres termes, il y a une différence entre considérer le saut d'instruction comme une cause ou une conséquence. C'est là encore une question de point de vue : est-ce que le modèle de saut d'instruction est obtenu à partir d'une caractérisation précise du processeur, ou est-il utilisé comme un modèle très général, comme le font bon nombre de méthodes d'analyse purement logicielles ?

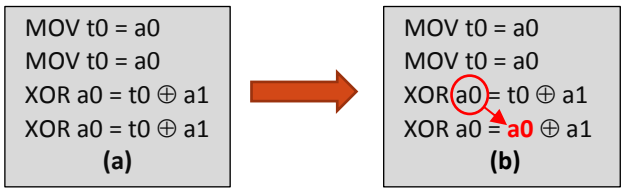

FIGURE 17 : REPRESENTATION DE L'ATTAQUE PAR FORWARDING SUR LA CONTREMESURE DE DUPLICATION SIMPLE

III.4.3.2. Duplication - Comparaison

Une contremesure de duplication et comparaison peut paraître plus efficace. La Figure 18 montre cette contremesure appliquée à une instruction de lecture mémoire. L'adresse de lecture est la somme d'un offset avec la valeur du registre assembleur s0.

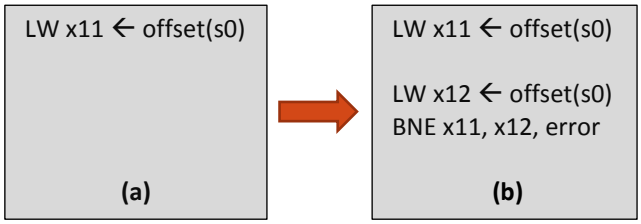

FIGURE 18 : (A) INSTRUCTION DE LECTURE MEMOIRE NON PROTEGEE ; (B) INSTRUCTION DE LECTURE MEMOIRE PROTEGEE DUPLICATION-COMPARAISON

Nous avons pu voir qu'il était possible d'utiliser le forwarding comme effet secondaire pour corrompre le résultat d'une instruction sans que cette corruption soit vue par la (ou les deux) instruction(s) suivante(s). En utilisant cette attaque, il est donc possible d'empêcher l'écriture du résultat de la lecture mémoire dans le registre assembleur **x12** ; sans que l'erreur soit détectée par la comparaison qui suit. Au niveau assembleur, nous pouvons représenter le comportement de cette attaque comme sur la Figure 19, avec une sauvegarde de la valeur de **x12**, puis une restauration juste après la comparaison. Nous pouvons considérer que cette attaque simple a un double effet : corrompre la lecture mémoire tout en contournant la contremesure.

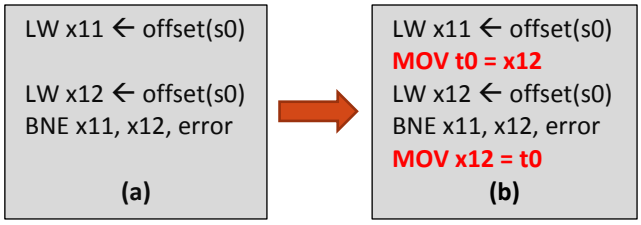

FIGURE 19 : REPRESENTATION DE L'ATTAQUE *REG_WRITE* SUR UNE INSTRUCTION DE LECTURE MEMOIRE PROTEGEE PAR DUPLICATION COMPARAISON. BNE (BRANCH IF NOT EQUAL) EST L'INSTRUCTION DE BRANCHEMENT COMPLEMENTAIRE DE BEQ. L'ATTAQUE EST MODELISEE PAR UNE SAUVEGARDE DE LA VALEUR DU REGISTRE ASSEMBLEUR A2, PUIS UNE RESTAURATION JUSTE APRES LA CONTREMESURE.

III.4.3.3. Duplication et double comparaison

Une manière de rendre la contremesure précédente plus efficace serait de dupliquer également la comparaison, comme sur la Figure 20.a. Cette évolution permettrait de détecter l'attaque sur la lecture mémoire car pour ce type d'instruction, le forwarding ne peut fonctionner que sur un cycle (on ne pourrait pas passer la seconde comparaison). Mais cela ne protègerait pas forcément d'autres types d'instructions comme les instructions de type R, qui peuvent utiliser le forwarding sur deux cycles.

D'autre part, il existe d'autres manières d'attaquer cette contremesure. Nous pouvons par exemple nous servir de la faute qui permet de modifier la valeur d'un registre pendant une instruction de branchement. En visant la dernière comparaison, nous pouvons donc remplacer la valeur du registre assembleur x12 par zéro, comme montré dans la Figure 20.b. Dans ce cas, c'est donc en attaquant directement la contremesure que l'on impacte la donnée protégée. Cette attaque n'est possible cependant que dans le cas où un registre multiple de 4 est utilisé.

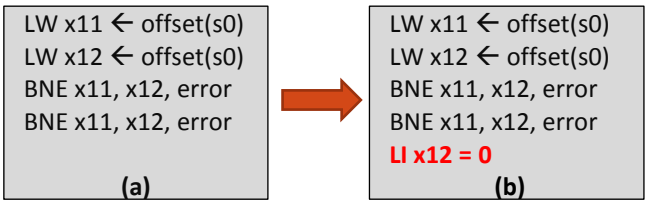

FIGURE 20 : REPRESENTATION DE L'ATTAQUE *REG_WRITE* POUR UNE INSTRUCTION DE LECTURE MEMOIRE PROTEGEE PAR DUPLICATION ET DOUBLE COMPARAISON. LA DERNIERE INSTRUCTION DE COMPARAISON PEUT POTENTIELLEMENT ECRIRE 0 DANS LE REGISTRE ASSEMBLEUR X12.

III.4.3.4. Triplication

L'avantage d'une contremesure de triplication est que plutôt que se limiter à détecter une faute, elle permet de la corriger. Au vu des comportements montrés en partie III.4.2.1, un attaquant pourrait cependant se servir de cette caractéristique pour produire de nouvelles attaques. Nous pouvons en particulier nous tourner vers les instructions sur la mémoire. Certaines fautes permettent de modifier la valeur contenue dans un endroit de la mémoire juste après avoir lu cette valeur, comme celles agissant sur le signal *Mem_cmd* pendant les instructions de lecture mémoire (voir Tableau 2). Si cette faute est injectée lors de la première instruction de lecture, les deux suivantes lisent la valeur corrompue, et après le vote de majorité, ce sera cette valeur corrompue qui sera utilisée pour le reste de l'exécution.

Cette attaque ne peut fonctionner que si les instructions seulement sont tripliquées et pas le contenu de la mémoire. Une triplication de la mémoire nécessiterait cependant de tripliquer les opérations d'écriture, or une opération d'écriture peut également modifier le registre assembleur qu'elle est sensée écrire.

Ces attaques sur la triplication, ainsi que les attaques sur la contremesure de duplication de Moro montrent que les capacités de correction de fautes ne sont pas forcément suffisantes dans un contexte de sécurité. Si cette capacité de correction a du sens pour des modèles de faute logiciels simples, la complexité réelle des comportements fautifs peut la rendre contre-productive en laissant s'exécuter des attaques qui auraient pu être détectées.

III.4.3.5. Intégrité du flot de contrôle

Il existe différents schémas de protection pour protéger le flot de contrôle. La plupart sont basés sur des comparaisons avec des signatures calculées à l'avance. Sur la Figure 21, est représentée la forme que peut prendre cette contremesure. Chaque bloc de base commence par une mise à jour de la signature et la vérification de cette signature avec une valeur calculée au moment de la compilation. A noter que la mise à jour de la signature peut également être effectuée à d'autres endroits du bloc, en fonction du schéma utilisé [64].

Comme nous avons pu le voir auparavant, dans le cas où il y a une mauvaise spéculation, il est possible de réactiver la troisième instruction fantôme après le branchement. Donc dans la Figure 21, il est possible d'exécuter l'instruction 13 avant de passer dans le bloc de gauche, ou d'exécuter l'instruction 3 avant de rentrer dans le bloc de droite. Ce comportement constitue donc une claire violation du flot de contrôle, qui n'est pas détectée par la contremesure.

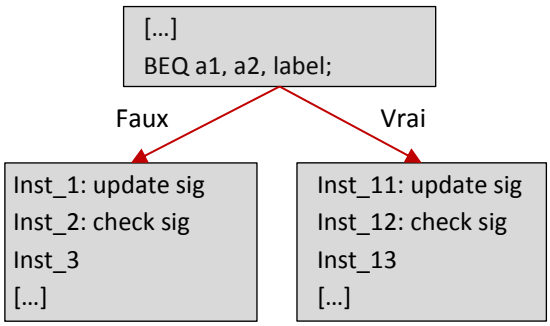

FIGURE 21 : REPRESENTATION D'UNE CONTREMESURE D'INTEGRITE DU FLOT DE CONTROLE

Dans le processeur étudié, il est possible de réaliser ce type d'attaque dans le cas où les processus de mise à jour et de vérification de signature ne prennent pas plus de deux instructions au début du bloc. L'attaque semble donc possible avec plusieurs schémas décrits dans [64] (« semble » car les schémas sont décrits au niveau source et qu'il y a donc la phase de compilation à prendre en compte), notamment les schémas RSCFC, SEDSR, SCFC et SIED.

III.4.4. Conséquences sur une application réelle

Les attaques précédentes ont été effectuées sur des contremesures isolées. Mais il est également possible de se servir des fautes complexes pour attaquer une application réelle. Dans cette section, nous montrons quelques exemples d'attaques sur une application de vérification de PIN sécurisée.

III.4.4.1. Présentation de l'application VerifyPIN

L'application, appelée VerifyPIN, est tirée de la bibliothèque FISSC [21]. Cette bibliothèque est constituée de différents programmes avec différents types de contremesures. En particulier, VerifyPIN existe en plusieurs versions avec un nombre croissant de contremesures, ce qui permet entre autre de tester l'efficacité de ces contremesures.

VerifyPIN est une application qui compare un code PIN entré par l'utilisateur (userPIN) à un code PIN secret (cardPIN). Ces codes PIN sont constitués de quatre chiffres (par défaut). L'authentification est réussie si tous les chiffres du code utilisateur correspondent à ceux du code secret. Le code de ce programme est présenté sur la Figure 22. Il est composé de deux fonctions : la première gère l'authentification et doit faire appel à la seconde pour comparer les codes PIN. Le programme a également un compteur d'essais, initialisé à trois et décrémenté à chaque tentative pour limiter le nombre de tentatives d'authentification.

VerifyPIN

```
authentification = 0;
if (cnt > 0){
   if(Compare() == 1){
      cnt=3 ;
      authentification = 1;
   } else cnt-- ;
}
```

Compare

```
for(i=0 ; i<4 ; i++){
   if(userPIN[i] != cardPIN[i])
      return 0;
}
return 1;
```

FIGURE 22 : EXTRAIT DU CODE DE VERIFYPIN NON PROTEGE. CERTAINES INITIALISATIONS NE SONT PAS REPRESENTEES

Dans cette section, nous considérerons la version 6 du programme, présentée en Figure 23, qui contient les contremesures suivantes :

- Booléens durcis : les valeurs des booléens sont fixés à 0x55 (booléen faux) et 0xAA (booléen vrai)
- Boucle de taille fixe : la boucle doit normalement s'exécuter quatre fois. A la fin de la boucle, une vérification est faite pour vérifier que l'itérateur est bien égal à quatre.
- Tests doublés : les tests conditionnels sont effectués deux fois et une alerte est levée si ces tests aboutissent à deux résultats différents.
- Expansion en ligne de la fonction *Compare* : le corps de la fonction *Compare* est inclus directement dans la fonction *VerifyPIN* pour éviter le processus d'appel de fonction.

VerifyPIN

```
1     authentification = FALSE;
2     if(cnt >= 0) {
3        cnt--;
4        status = FALSE;  diff = FALSE;
5
6        for(i = 0 ; i < 4 ; i++) {
7           if(userPin[i] != cardPin[i])  diff = TRUE;
8        }
9
10       if(i != 4) attaque_detectee();
11
12       if (diff == FALSE) {
13          if(FALSE == diff)  status = TRUE;
14          else attaque_detectee();
15       } else status = FALSE;
16
17       if(status == TRUE) {
18          if(TRUE == status) {
19             cnt = 3;
20             authentification = TRUE;
21          } else attaque_detectee();
22       }
23    }
```

FIGURE 23 : EXTRAIT DU CODE DE VERIFYPIN PROTEGE

La compilation du programme est un élément important à prendre en compte. Le compilateur utilisé est GCC. Sans aucune optimisation (-O0), l'allocation de registres est très mauvaise et le fichier objet est constitué d'énormément d'accès mémoire. D'un autre côté, le premier niveau d'optimisation (-O1) supprime certaines contremesures. Pour avoir un binaire avec une allocation de registre correcte tout en conservant les contremesures, le niveau d'optimisation de debug (-Og) a finalement été retenu. Ce niveau d'optimisation permet de conserver la structure du code source.

III.4.4.2. Première attaque : Authentification par modification du forwarding

Une première attaque consiste à exploiter la contremesure des booléens durcis. Trois spécificités doivent être prises en compte :

- Les booléens durcis utilisent les valeurs 0x55 et 0xAA. Comme discuté en partie II.3.3.3, grâce à ces valeurs, il paraît compliqué de passer simplement d'une valeur à l'autre en injectant des fautes. Cependant, nous pouvons quand même remarquer que 0x55 + 0x55 est égal à 0xAA. En d'autres termes, additionner deux booléens faux donne un booléen vrai. Bien sûr, à ce stade, il n'y a pas encore de raison de penser qu'on puisse additionner des booléens…
- Dans une architecture RISC-V, il n'y a pas d'instruction pour assigner directement une valeur à un registre assembleur. L'assignation se fait au moyen d'une addition avec le registre assembleur zero (qui est, pour rappel, fixé à la valeur 0). Par exemple mettre le registre assembleur a5 à FAUX est implémenté de la manière suivante : « ADDI a5 = zero + 0x55 »
- Dans l'implémentation du processeur utilisé, l'utilisation du registre assembleur zero utilise toujours la structure de forwarding (le forwarding est donc activé).

En exploitant ces différentes spécificités, il est possible de monter une attaque sur la partie du programme qui teste la valeur de la variable *diff* après la boucle, pour savoir si les codes PIN ont une

différence ou non. Cette partie du programme est représentée dans la Figure 24. La variable *diff*, contenue dans le registre assembleur s0, doit être comparée à 0x55, donc cette valeur est d'abord chargée dans le registre assembleur a5. Si *diff* n'est pas égale à 0x55, c'est qu'il y a une différence entre les codes PIN, donc que l'authentification a échoué. Cet échec consiste à mettre la variable *status*, dans le registre assembleur a5, à FAUX.

ADDI a5 = zero + 0x55 **BNE** s0, a5, offset […] **ADDI** a5 = zero + 0x55	Met a5 à FAUX pour la comparaison Compare s0 (*diff*) à a5 (FAUX) Met *status* à FAUX

FIGURE 24 : PARTIE DU CODE ASSEMBLEUR DE VERIFYPIN

Dans le cas où la prédiction de branchement est correcte, la première instruction ADDI est toujours dans le pipeline quand la seconde est exécutée. Cette deuxième instruction ADDI utilise le forwarding pour utiliser le registre zero. En injectant une faute simple-bit, il est possible de modifier quelle valeur est forwardée, et en particulier de transmettre à la place de zero le résultat de la première instruction ADDI. Le calcul de « zero + 0x55 » est ainsi remplacé par « 0x55 + 0x55 », soit 0xAA. La variable *status* qui devait avoir la valeur FAUX se retrouve donc avec la valeur VRAI, ce qui conduit à une authentification réussie. La Figure 25 représente cette attaque.

ADDI a5 = zero + 0x55 **BNE** s0, a5, offset […] **ADDI** a5 = 0x55 + 0x55	Forward du résultat de l'avant-dernière instruction exécutée

FIGURE 25 : REPRESENTATION DE L'ATTAQUE PAR MODIFICATION DU FORWARDING

Ainsi, la seule condition pour réussir cette attaque est que la spéculation soit correcte. A rebours des discussions sur l'exécution spéculative, c'est ici une prédiction incorrecte qui permettrait d'éviter une vulnérabilité.

Pour contrer cette attaque de manière plus certaine, deux possibilités peuvent être envisagées. L'attaque est rendue possible du fait de la relation existant entre les valeurs FAUSSE et VRAIE des booléens durcis. Il peut donc être plus intéressant de se tourner vers des valeurs qui n'ont a priori pas de relation, comme 0xC5 et 0xA3. Une seconde possibilité est de lutter contre l'optimisation de forwarding en insérant une instruction nulle avant la seconde instruction ADDI. Ainsi, la première instruction ADDI a le temps de terminer son exécution et son résultat ne peut plus être forwardé.

III.4.4.3. Deuxième attaque : Authentification par réutilisation du multiplicateur

Nous avons vu dans une partie précédente, que certains registres pouvaient conserver leur résultat pendant un certain temps. C'est le cas de la sortie de la structure de multiplication/division. Celle-ci reste disponible dans un registre caché jusqu'à ce qu'elle soit écrasée par une nouvelle multiplication/division. En injectant une faute précise, ce résultat peut être réinjecté dans le banc de registres.

Dans l'hypothèse où un attaquant a la possibilité d'effectuer des opérations avant l'exécution de VerifyPIN (par exemple dans un système où certaines opérations peuvent être effectuées par n'importe quel utilisateur, mais d'autres nécessitent un code PIN), celui-ci peut exécuter une multiplication dont le résultat est 0xAA. L'attaquant met ainsi certaines structures du processeur

dans un état connu. Pendant l'exécution de VerifyPIN, ce résultat peut être réutilisé pour, par exemple, forcer la valeur de la variable *status* et ainsi s'authentifier.

Pour protéger contre ce type d'attaque, il est nécessaire d'effacer le résultat contenu dans le registre caché, ce qui revient dans ce cas à exécuter une multiplication (dont le résultat ne peut pas être un problème dans VerifyPIN).

L'aspect intéressant de cette attaque est que par l'intermédiaire de registres cachés, il est possible de transmettre des valeurs en provenance de l'extérieur du programme, pendant l'exécution du programme.

III.4.4.4. Troisième attaque : Safe-error sur les chiffres du code secret

Cette troisième attaque est une attaque de type safe-error, qui consiste donc à déduire des informations sur le code secret de l'application à partir de son comportement macroscopique. Il n'est pas possible ici d'exploiter la sortie de l'application car c'est une valeur booléenne (authentification ou non). Mais l'application verifyPIN peut également déclencher une alerte si une faute est détectée. C'est le cas par exemple si l'itérateur de la boucle n'est pas égal à 4 à la fin de la boucle. L'action engendrée par cette alerte n'est pas fixée, mais plusieurs possibilités peuvent être envisagées, qui vont d'un simple timeout à une autodestruction du circuit. Nous poserons comme hypothèse que l'attaquant peut continuer à utiliser le circuit même après avoir déclenché la contremesure (mais en recommençant le processus d'authentification).

Le déclenchement ou non de cette alerte peut servir à faire fuiter de l'information sur le code secret. En d'autres termes, le but de la faute injectée est de conditionner le déclenchement de l'alerte à la valeur du secret. Plus précisément, l'attaque présentée ici permet de savoir si un chiffre du code secret est dans l'intervalle {0..3} ou {4..9}. En injectant une faute précise, l'alerte est déclenchée dans le second cas, mais pas dans le premier. La Figure 26 présente la partie du code assembleur qui nous intéresse ici : celle qui concerne la boucle de comparaison des PIN.

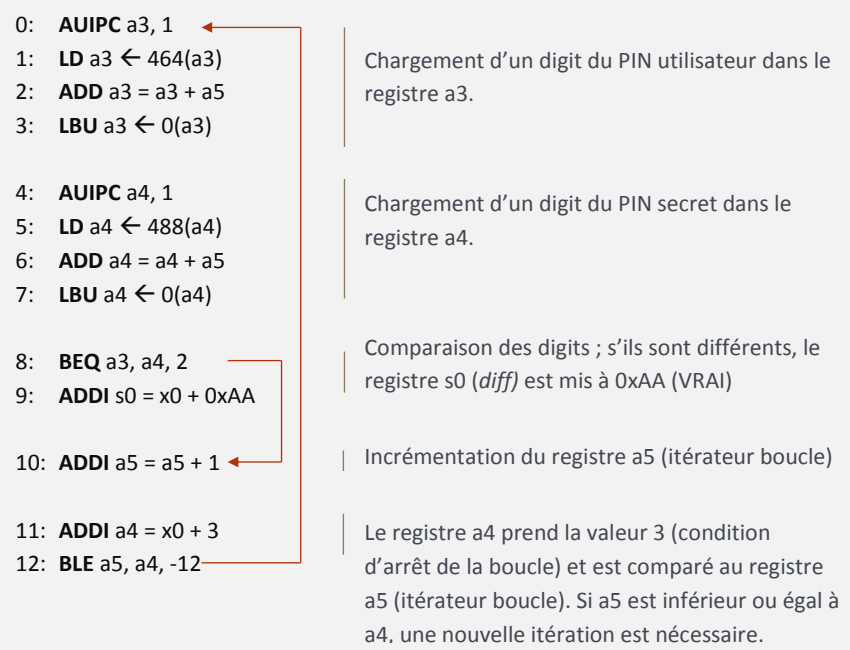

FIGURE 26 : CODE ASSEMBLEUR DE LA BOUCLE DE COMPARAISON DES PIN. LES INSTRUCTIONS LD ET LBU SONT DES INSTRUCTIONS DE LECTURE MEMOIRE ; L'INSTRUCTION AUIPC PERMET DE CREER UNE VALEUR EN FONCTION DU COMPTEUR DE PROGRAMME, UTILISEE ICI POUR CALCULER UNE ADRESSE MEMOIRE

L'attaque vise le test de l'itérateur de la boucle de vérification des digits du PIN utilisateur (ligne 12). Si lors de ce test, l'itérateur de boucle est inférieur ou égal à 3, une nouvelle itération est requise (car il faut quatre tours de boucle pour vérifier les quatre digits du PIN). En analysant les instructions précédentes, nous pouvons déduire que le deuxième argument de la comparaison (le registre a4) est forwardé car sa valeur provient de l'instruction précédente (ligne 11). Une des attaques possibles est une désactivation de ce forwarding. Dans ce cas, plutôt que la valeur 3, la valeur utilisée dans la comparaison est une concaténation de la valeur du registre caché MSB et du mot de deux bits « 10 ». La comparaison devient donc : est-ce que « MSB | 0b10 ≤ 3 » ? (où '|' représente ici la concaténation).

Or, en remontant le programme, il est possible de déduire que la valeur de MSB au moment de cette comparaison contient les 62 bits de poids fort du digit du code secret qui a été examiné pendant l'itération courante. Cette valeur est chargée au moment où l'instruction de comparaison à la ligne 8 est exécutée et est conservée puisqu'aucune des instructions suivantes n'a besoin d'un second argument venant du banc de registre. Dans le cas où le chiffre secret est dans l'intervalle {0..3}, MSB est donc égal à 0, la comparaison est vraie, donc la boucle continue. Dans le cas où le chiffre secret est dans l'intervalle {4..9}, MSB est différent de 0, donc la valeur concaténée est strictement supérieure à 3 ; la comparaison est fausse, donc la boucle s'arrête. Pour résumer, suivant la valeur du chiffre secret, il est donc possible de sortir prématurément ou tardivement de la boucle, ce qui déclenche ensuite l'alerte.

Protéger contre cette attaque est très simple : il suffit d'inverser l'ordre des arguments de la comparaison. Dans le code de l'application, au lieu de comparer le chiffre entré par l'utilisateur avec le chiffre secret, il vaut mieux comparer le chiffre secret au chiffre entré par l'utilisateur. Ainsi, la même attaque ne ferait que révéler de l'information sur le chiffre entré par l'attaquant lui-même, ce qui est inutile.

Comme l'application VerifyPIN possède un compteur d'essais qui se décrémente à chaque essai, il est clair que cette attaque est difficilement exploitable en réalité, à moins d'avoir une manière de réinitialiser le compteur d'essai. L'intérêt de cette attaque est surtout de montrer une nouvelle fois comment les registres internes peuvent être exploités pour faire fuiter des données potentiellement secrètes du programme.

III.4.4.5. Quatrième attaque : Fuite du code secret complet

Pour cette dernière attaque, il est nécessaire de considérer la façon dont la fonction *verifyPIN* est appelée dans la fonction *main*. L'expansion en ligne peut être un moyen d'augmenter la sécurité en rendant plus difficile une attaque qui consisterait à sauter l'exécution de cette fonction. C'est aussi une façon d'optimiser le programme en s'affranchissant de la procédure d'appel.

Cependant, l'expansion en ligne d'une fonction enlève aussi les « frontières » entre les fonctions, matérialisées par les procédures d'appel et de retour, ce qui rend plus facile le fait de transmettre des données d'une fonction vers l'extérieur de cette fonction, par l'intermédiaire de registres cachés. En effet, ces procédures d'appel et de retour font intervenir notamment une sauvegarde puis une restauration du contexte, qui modifient l'état du processeur et effacent potentiellement des données sensibles des registres cachés. En l'occurrence, si l'expansion en ligne est utilisée pour la fonction *verifyPIN* dans la fonction *main,* il est possible de faire fuiter le code secret, par le moyen décrit ci-après.

Dans la partie III.4.2.2, nous avons vu que la dernière donnée chargée depuis la mémoire reste disponible sur une des voies du forwarding. Or, dans verifyPIN, c'est le code secret qui est lu en dernier en mémoire, donc la valeur de ce code reste disponible dans le forwarding même après la fin de la fonction. Suivant ce qui suit la fonction, il est donc possible de révéler ce code secret, en utilisant une attaque d'activation du forwarding ou de modification de l'étage forwardé. Cette attaque reste possible tant qu'il n'y a pas de nouvelle lecture mémoire ; c'est pourquoi le fait de ne pas utiliser l'expansion en ligne protègerait le programme : la procédure de retour utilise un appel mémoire. L'expansion en ligne peut être dangereuse lorsqu'on considère les registres cachés, car cela réduit l'indépendance entre les fonctions. Il est intéressant de constater que D'Silva et al. [19] montrent que l'expansion en ligne peut aussi poser des problèmes similaires en considérant la pile d'exécution.

Cette vulnérabilité est d'autant plus problématique que dans le processeur attaqué, c'est l'ensemble du code secret qui est révélé et pas seulement le dernier chiffre. En effet, dans le programme, le code est stocké dans un tableau de caractères (variables de un octet). Donc ce code secret est stocké dans des adresses contiguës en mémoire. Or la donnée qui peut être forwardée est un mot complet de 64 bits présent à l'adresse de lecture (aligné sur 8 octets). C'est donc bien l'ensemble du code secret (8 octets, dont les quatre premiers sont le code secret) qui est fuité et pas seulement le dernier chiffre (1 octet). La Figure 27 montre le chronogramme de la valeur qui peut être forwardée, durant les deux dernières itérations du programme.

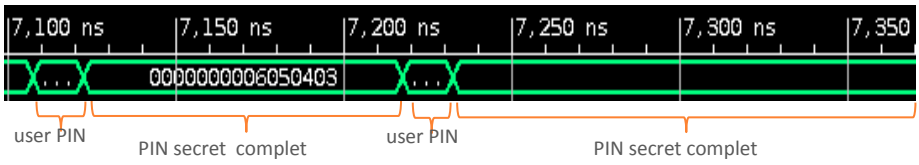

FIGURE 27 : CHRONOGRAMME MONTRANT LA VALEUR D'UN REGISTRE CACHE POUVANT ETRE FORWARDE EN FORÇANT LE FORWARDING. LE CODE SECRET EST 3456, VISIBLE QUAND LE SIGNAL VAUT 0X0000000006050403. CELUI-CI RESTE DISPONIBLE PENDANT DE LONGUES PERIODES DE TEMPS.

Outre le fait de ne pas utiliser l'expansion en ligne, il existe un autre moyen très simple de protéger contre cette attaque. Comme dans l'attaque précédente, il suffit d'inverser les arguments de la comparaison des PIN, de façon à d'abord charger la valeur secrète, puis la valeur utilisateur. La valeur qui restera dans le registre caché sera ainsi le code utilisateur. Une autre considération, pour limiter la quantité d'information fuitée, est d'utiliser un tableau de variables de 64 bits plutôt que de 8 bits. Ainsi, les différents chiffres du code secret seront stockés dans des mots différents en mémoire.

III.4.4.6. Conclusion sur les attaques de VerifyPIN

Ces différentes attaques, toutes validées dans des simulations RTL, permettent de montrer qu'il est possible de trouver des injections de fautes précises qui permettent de passer outre les contremesures logicielles d'une application sécurisée. Bien que les attaques présentées ne soient réalisables que dans des circonstances bien précises, il n'empêche qu'elles montrent les possibilités offertes par les attaques en faute notamment dans les registres cachés. Dans un contexte de sécurité, chaque vulnérabilité compte ; d'où la nécessité de prendre en compte les spécificités de l'architecture pour protéger un programme. Il a par ailleurs été montré que dans certains cas, il

pouvait être très facile et très peu coûteux d'éviter des vulnérabilités, comme le fait d'utiliser la donnée secrète comme premier argument dans une comparaison.

La discussion sur la difficulté à réaliser de telles attaques suppose de définir des hypothèses quant au pouvoir de l'attaquant, ce qui n'est pas l'objet de cette thèse. Le but n'est pas de donner un verdict sur la sécurité d'un système ; nous montrons seulement certains types de comportements qui peuvent être provoqués. Au développeur ensuite de trancher s'il considère l'attaque réalisable ou non.

Nous avons pu mettre en avant dans ce chapitre le rôle joué par les registres cachés, qui peuvent temporairement stocker des valeurs du programme. Pour se prémunir de ces attaques, quelques moyens simples ont été exposés. Toutefois, ces moyens sont très spécifiques. Pour protéger de manière plus générale, une autre approche est de faire en sorte de vider ces registres cachés pour qu'ils ne contiennent plus de données sensibles. Cette approche est similaire à celle déjà utilisée pour éviter des fuites dans la mémoire RAM (il ne faut pas qu'à la fin d'un algorithme de cryptographie, la clé soit toujours présente dans le cache par exemple). Dans ce cas, des écritures sont effectuées dans la mémoire pour remplacer les valeurs qui y ont transité. Dans le cas des registres cachés, ce processus est cependant plus complexe car il n'existe pas d'instruction pour remplacer directement le contenu de ces registres. Il est donc nécessaire d'utiliser des effets de bord des instructions. Par exemple, pour remplacer la valeur du registre MSB du deuxième argument de l'ALU, il faut exécuter une instruction de type R, en faisant attention à ne pas déclencher le forwarding. Une fois encore, seule une bonne compréhension de la microarchitecture du processeur peut permettre de localiser ces registres cachés et savoir quelle séquence d'instructions peut permettre de les vider.

III.4.5. Conclusion

Cette partie a permis de mettre en évidence divers comportements qu'il est possible de provoquer dans notre processeur. Ces comportements ont ensuite pu être mis à profit pour attaquer des contremesures typiques et une application sécurisée de vérification de code PIN.

Nous pouvons voir que le décalage entre la vision logicielle et la réalité matérielle, évoqué dans la partie précédente, peut avoir des conséquences concrètes sur la sécurité des applications. Le forwarding a notamment pu être exploité à plusieurs reprises et les moyens de protéger sont souvent de lutter contre les optimisations du processeur.

Si certains comportements présentés pourraient être considérés comme faisant partie des modèles typiques, cette affirmation est à nuancer. En simplifiant un peu, la première attaque de VerifyPIN consiste à remplacer la valeur 0 par la valeur 0x55. Ce comportement est donc apparenté à une simple corruption de registre. Pourtant, une telle corruption serait bien souvent jugée improbable car très difficile à mettre en œuvre. Mais en exploitant intelligemment l'architecture du processeur, une injection dans une seule bascule suffit à obtenir ce comportement. Les comportements montrés ici, et en particulier ceux mettant en jeu le forwarding, n'utilisent pas des valeurs fixes, mais permettent de réutiliser des données ayant été utilisées précédemment dans le programme. Cette nuance est importante car elle montre l'importance du lien matériel-logiciel.

Le but de cette partie était également de donner un aperçu de la complexité induite par ces comportements. Il y a une profusion d'effets très différents les uns des autres et il semble que l'on

ne puisse se limiter à un seul modèle de faute logiciel pour représenter tous les comportements fautifs possibles.

Avant de clore cette partie, il est utile de discuter du fait que le processeur utilisé dans cette étude ne possède pas de contremesures matérielles et que la présence de telles contremesures pourrait rendre impossible certaines des attaques présentées. Il est vrai qu'un processeur protégé au niveau matériel présenterait probablement moins de comportements exploitables ; pourtant, cela n'invalide pas le principe d'une étude de l'architecture d'un processeur, et ce pour plusieurs raisons. D'abord parce qu'un processeur n'est jamais sécurisé à 100% ; il existera toujours des points plus vulnérables que seule une étude approfondie de l'architecture pourrait révéler. Ensuite parce que les structures qui sont protégées en priorité dans un processeur sont souvent le banc de registres ou la mémoire car ce sont des structures qui représentent une grande surface du processeur et qu'il est relativement facile de protéger ; hors ces structures ont été exclues de notre étude : les fautes étaient uniquement injectées dans le cœur du processeur, moins souvent protégé. Enfin, comme nous l'avons vu précédemment, les contremesures matérielles peuvent être relativement coûteuses. Il est donc intéressant d'étudier d'abord le processeur non protégé et voir s'il est possible de sécuriser le système directement au niveau logiciel, ce qui se révèlerait moins coûteux.

III.5. Conclusion du chapitre

Ce chapitre aura permis d'explorer les conséquences qu'ont diverses optimisations du processeur sur l'analyse de sécurité. Ces optimisations créent un large fossé entre la vision logicielle de la sécurité et la réalité matérielle. Ces différences sont à l'origine de nombreux phénomènes complexes qu'il est possible de provoquer dans un processeur en injectant des fautes précises. Le seul fait d'utiliser un pipeline brise la vision logicielle d'atomicité des instructions, qui conduit à des phénomènes tels qu'une faute impactant le résultat d'une instruction sans que cet impact soit vu par les instructions suivantes.

Après avoir étudié l'implémentation de ces optimisations dans un processeur d'architecture RISC-V, des injections de fautes dans des simulations RTL ont permis de mettre en lumière divers comportements du processeur. Ces comportements montrent par exemple que certaines valeurs ou registres ont un impact plus grand que d'autres, que l'ordre des arguments d'une instruction a une importance, ou encore que le contexte dans lequel est exécutée une instruction joue également un rôle. Ces comportements provoqués par les fautes ont ensuite pu être exploités pour monter diverses attaques sur des contremesures logicielles typiques et sur une application de vérification de PIN sécurisée.

Après cette étude spécifique de quelques comportements fautifs, se pose la question d'une modélisation plus globale et plus automatisée des effets des fautes au niveau logiciel. Ceci suppose notamment d'avoir un moyen de représenter les effets des fautes au niveau logiciel, ainsi qu'une méthode pour évaluer la pertinence de cette représentation par rapport aux effets réels. Le développement d'une telle approche est l'objet du chapitre suivant.

Chapitre IV. Liaison entre injection de faute matérielle et logicielle

Résumé du chapitre

Jusqu'à présent, des fautes étaient injectées et analysées au cas par cas. Pour modéliser plus précisément leurs effets possibles dans un processeur, il est nécessaire d'adopter une approche plus systématique.

Dans ce quatrième chapitre, nous développons une approche mettant en jeu des injections au niveau RTL et au niveau logiciel. Le but de cette approche est de faire converger les résultats obtenus dans ces deux niveaux d'abstraction. Pour réaliser les injections au niveau logiciel, à partir de modèles représentant des effets semblables à ceux décrits dans le chapitre précédent, un outil de mutation de programme est développé. Des métriques et des méthodes d'analyse sont ensuite introduites pour étudier le lien entre injection de faute RTL et injection de faute logicielle.

Ces méthodes d'analyse sont, pour finir, appliquées à l'analyse des comportements fautifs d'un processeur d'architecture RISC-V.

Sommaire

IV.1. Introduction du chapitre

Dans le chapitre précédent, nous avons montré quelques effets complexes qu'il est possible d'obtenir en injectant des fautes dans des structures spécifiques du processeur. Cette étude était « manuelle », dans le sens où les circonstances de chaque injection de faute étaient spécifiées précisément. Pour avoir une vue plus exhaustive des comportements possibles, il est nécessaire d'automatiser au moins en partie la méthode.

Le but de l'approche développée dans ce chapitre est d'éclaircir la relation entre les niveaux d'abstraction RTL et logiciel. Nous pouvons voir cette relation comme une relation à double sens, illustrée en Figure 28. Le passage du niveau RTL au niveau logiciel est un processus d'abstraction : le but est de modéliser au niveau logiciel ce qu'il se passe au niveau matériel (RTL). En sens inverse, nous trouvons un processus de concrétisation qui consiste à lier les modèles logiciels à des structures spécifiques du processeur, afin par exemple de réaliser en simulation RTL des fautes prédites par les modèles logiciels. Cet aspect de concrétisation est rarement pris en compte dans la littérature car l'objectif est souvent la modélisation logicielle ; il est pourtant intéressant car, comme nous le verrons, il peut permettre de prendre de meilleures décisions quant au choix des contremesures matérielles et logicielles.

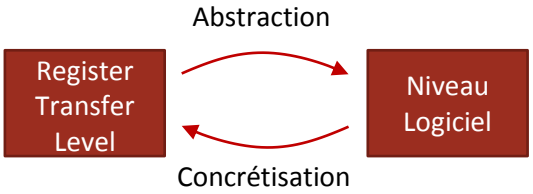

FIGURE 28: RELATION A DOUBLE SENS ENTRE RTL ET NIVEAU LOGICIEL

Pour étudier cette relation à double sens, une méthode a été développée, basée sur des comparaisons entre injections RTL et injections au niveau logiciel, dans des circonstances bien maîtrisées pour pouvoir conclure de façon précise.

Nous commencerons ce chapitre par une vue d'ensemble de la méthode utilisée. Dans un deuxième temps, comme la méthode est basée sur la comparaison d'injections, nous détaillerons le déroulement des injections au niveau RTL et au niveau logiciel. Puis, nous pourrons définir plusieurs méthodes d'analyse pour évaluer les processus d'abstraction et de concrétisation. Enfin, nous appliquerons ces méthodes pour étudier et modéliser de manière précise les comportements fautifs d'un processeur d'architecture RISC-V.

IV.2. Vue d'ensemble de l'approche

IV.2.1. Objectifs de l'approche

Le premier objectif de l'approche est de modéliser des comportements fautifs du processeur au niveau logiciel. Cela correspond au processus d'abstraction. A cet égard, la comparaison de résultats d'injection aux niveaux RTL et logiciel permet de calculer plusieurs métriques pour quantifier la pertinence de la modélisation logicielle. Nous en développons ici deux : une métrique de couverture et une métrique de justesse. La métrique de couverture montre la proportion d'injections RTL qui sont prédites par les modèles de faute logiciels. La métrique de justesse fait l'opération inverse : elle quantifie la proportion d'injections logicielles qui correspondent à des effets réels obtenus au niveau RTL. Cette seconde métrique peut être intéressante car la modélisation de fautes au niveau logiciel entraîne globalement une perte de précision ; les effets prédits par les modèles logiciels peuvent donner des faux positifs. Outre le calcul de ces métriques, la comparaison automatique des injections RTL et logicielles permet également d'autres analyses, comme l'identification rapide des comportements non couverts qui sont les plus fréquents ; ceci afin de guider le travail de modélisation.

Avec ces métriques, la modélisation des fautes au niveau logiciel peut être vue comme un processus itératif, montré en Figure 29.

Ce processus est divisé en plusieurs étapes :

1. Spécification de programmes de caractérisation dans lesquels les fautes seront injectées
2. Spécification des bascules à cibler pendant les injections RTL
3. Réalisation des injections RTL pendant l'exécution des programmes de caractérisation
4. Spécification de modèles de faute logiciels en étudiant la propagation des fautes RTL dans la microarchitecture du processeur
5. Réalisation des injections logicielles dans les programmes de caractérisation, avec les modèles de faute logiciels.
6. Comparaison et analyse des résultats des injections RTL et logicielles pour notamment calculer les métriques de couverture et de justesse.
7. Si les métriques ne sont pas satisfaisantes, retour à l'étape 4.

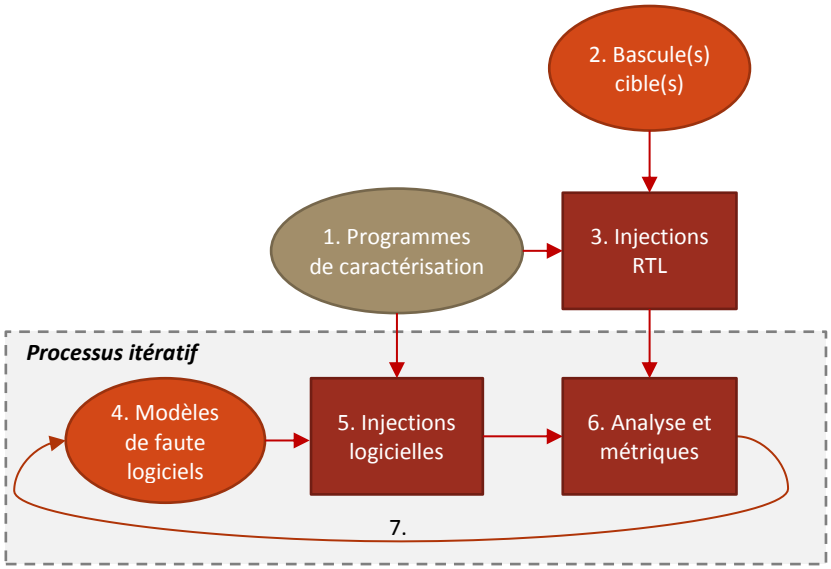

FIGURE 29: VUE D'ENSEMBLE DE L'APPROCHE

Le second objectif de l'approche est d'aider le processus de concrétisation. Celui-ci a deux intérêts. Le premier est de pouvoir réaliser au niveau RTL les effets prédits par un modèle logiciel. En effet, comme les modèles de faute ne sont pas fiables à 100%, il peut être intéressant de vérifier rapidement si une faute prédite par un modèle est faisable ou non en réalité. Pour cela, il est possible de lier les modèles de faute logiciels aux paramètres d'injection les plus probables ; en l'occurrence aux bascules qui peuvent le plus probablement créer l'effet désiré. Le second intérêt est d'aider certaines décisions au niveau matériel, comme le choix de contremesures. Il est en effet possible de savoir quelles sont les structures du processeur les mieux modélisées au niveau logiciel, donc celles dont la sécurité peut être évaluée au niveau logiciel et celles au contraire qui sont plus compliquées à modéliser et qu'il serait plus sage de protéger au niveau matériel.

Les différents types d'analyse esquissés ici seront explorés dans ce chapitre. Mais comme toute l'approche repose sur la comparaison d'injections au niveau RTL avec des injections au niveau logiciel, il nous faut commencer par définir précisément le processus expérimental des injections RTL et logicielles. Nous commençons dans les sections suivantes par présenter les circonstances des injections, communes aux deux niveaux d'abstraction. Sont définis les programmes dans lesquels les injections sont réalisées, ainsi que les instants d'injection et d'observation des résultats.

IV.2.2. Programmes de caractérisation

Les résultats d'une campagne d'injection de fautes sont nécessairement dépendants du programme dans lequel sont injectées les fautes. Pour réduire l'influence de ce biais et concevoir des modèles de faute qui représentent plus généralement ce qu'il se passe dans le processeur, il faut donc une certaine diversité dans les programmes utilisés. Nous avons d'ailleurs pu remarquer dans le chapitre précédent que certains comportements ne peuvent être provoqués que dans des conditions bien particulières. Si les programmes de caractérisation n'étaient pas suffisamment diversifiés, ces situations ne seraient pas forcément découvertes. Dans notre approche, nous utilisons une multitude de courts programmes, appelés programmes de caractérisation, qui représentent des situations variées.

Les programmes de caractérisation sont écrits au niveau assembleur et sont découpés en trois parties : le prologue, la cible et l'épilogue. La cible est l'instruction dans laquelle est injectée la faute

et le prologue et l'épilogue sont des séquences d'instructions placées avant et après la cible, qui représentent divers types de situations. La structure d'un programme de caractérisation est illustrée en Figure 30.

```
Prologue_instruction_1
Prologue_instruction_2
Cible                    // Injection
Epilogue_instruction_1
Epilogue_instruction_2
Epilogue_instruction_3
```

FIGURE 30: STRUCTURE D'UN PROGRAMME DE CARACTERISATION

Ce choix de découper les programmes de caractérisation en trois parties a été fait pour réduire la dépendance des résultats à ces programmes. Modifier le prologue sans toucher à la cible ou à l'épilogue permet de modifier l'état interne du processeur juste avant l'injection de faute, afin de générer des comportements différents (par exemple en faisant forwarder ou non les arguments à l'instruction cible). Modifier la cible sans toucher le prologue ou l'épilogue permet de tester dans les mêmes conditions divers types d'instructions. Enfin, modifier l'épilogue sans toucher au prologue ou à la cible permet de modifier la façon dont la faute se propage dans le processeur. En effet, certains effets latents du processeur peuvent se propager ou non en fonction des instructions qui suivent l'injection. Dans [15], une des catégories dans lesquelles sont classés les résultats d'injection est constituée des injections qui n'ont aucun effet visible mais pour lesquelles l'état interne final du processeur est différent par rapport à une exécution non fautée. En faisant varier l'épilogue de nos programmes de caractérisation, le but est justement de propager ces états latents afin de voir s'ils produisent au final des effets visibles.

IV.2.3. Observation des résultats

Un autre aspect clé d'une méthodologie d'injection de fautes réside dans la définition des signaux observés et du moment d'observation. Comme le but est de comparer des injections à différents niveaux d'abstraction, il est nécessaire de prendre en compte les contraintes liées à chaque niveau. Pour avoir un point de comparaison valable, il est nécessaire d'observer les mêmes choses au même moment.

En ce qui concerne l'objet de l'observation, la vision logicielle est assez restreinte : seules certaines parties de l'architecture matérielle sont visibles au niveau logiciel. Nous nous bornerons donc à observer le banc de registres et la mémoire, que ce soit pour les injections RTL ou logicielles. La seule autre structure que l'on pourrait aussi considérer au niveau logiciel serait le compteur de programme, mais celui-ci est déjà implicitement utilisé car les observations se font au même stade de l'exécution du programme : les résultats ne sont observés que si le compteur de programme a la valeur spécifiée.

Pour cette question du moment d'observation, la partie matérielle est plus complexe à gérer à cause des effets latents. Au niveau logiciel, spécifier l'état du banc de registres et de la mémoire à un moment donné du programme est suffisant pour complètement définir l'état de l'exécution (ceci parce qu'à ce niveau, les instructions sont considérées atomiques). Au niveau matériel en revanche, l'état du processeur entre également en jeu. Certains effets latents peuvent mettre plusieurs cycles d'horloge à se manifester de manière visible. Ainsi, observer les effets de la faute trop tôt signifie que certains effets de la faute n'auront pas le temps de se propager. Mais au contraire, observer trop

tard signifie que les effets de la faute peuvent être masqués par les instructions exécutées après l'injection. Dans le but de réduire les approximations associées à chaque option, il a été décidé de définir deux points d'observation. La première observation concerne les effets instantanés de la faute, c'est-à-dire les effets juste après l'exécution de l'instruction cible. La seconde observation concerne les effets de propagation de la faute ; elle est effectuée juste après l'exécution de la dernière instruction de l'épilogue. Ces points d'observation sont représentés sur la Figure 31.

```
Prologue_instruction_1
Prologue_instruction_2
Cible                    // Première observation après exécution de ceci
Epilogue_instruction_1
Epilogue_instruction_2
Epilogue_instruction_3   // Seconde observation après exécution de ceci
```

FIGURE 31 : POINTS D'OBSERVATION DANS UN PROGRAMME DE CARACTERISATION

Ces points d'observation sont faciles à définir du côté logiciel car les instructions sont considérées atomiques. Du côté des simulations RTL, ces points d'observation correspondent à un cycle d'horloge après que l'instruction associée soit passée dans le dernier étage du pipeline.

Pour résumer, en ce qui concerne l'observation des résultats d'injection, seuls le banc de registres et la mémoire sont considérés ; et ils sont observés à deux instants pour avoir à la fois les effets instantanés et les effets de propagation de la faute.

IV.3. Injection de faute au niveau RTL

Maintenant que les conditions d'injection et d'observation sont définies, il nous reste à exposer comment ces injections sont mises en œuvre, tout d'abord du côté RTL. Une vue d'ensemble des injections RTL est présentée sur la Figure 32 ; elle correspond au fonctionnement du bloc 3 de la Figure 29.

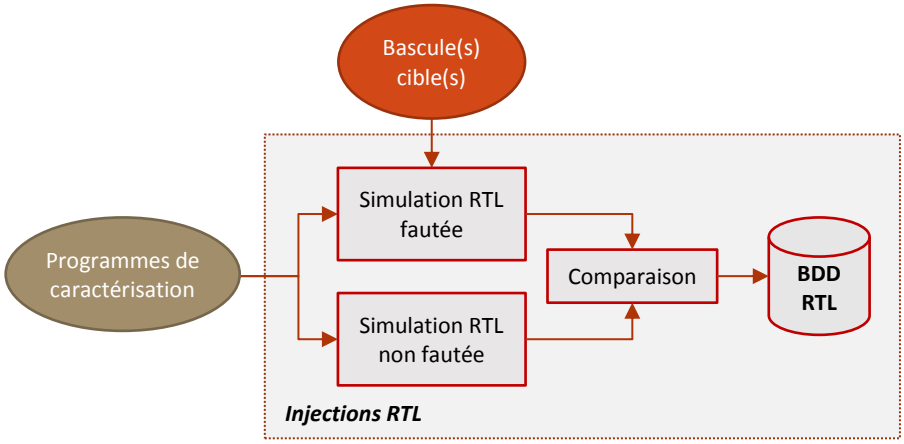

FIGURE 32 : VUE D'ENSEMBLE DE L'INJECTION RTL. LES PROGRAMMES DE CARACTERISATION SONT UTILISES DANS DES SIMULATIONS FAUTEES ET NON FAUTEE. LES RESULTATS SONT COMPARES ET STOCKES DANS UNE BASE DE DONNEES

IV.3.1. Vue d'ensemble

Comme dans le chapitre précédent, les injections sont effectuées en simulation RTL dans le logiciel QuestaSim, avec le modèle du bit-flip. Les injections sont simples ou multiples (définies plus loin). Les bascules ciblées sont celles du pipeline du processeur, en excluant celles qui font partie du banc de registres et de la mémoire. En effet, les effets des fautes dans ces deux structures sont plus

facilement prédictibles et nous sommes plutôt intéressés par les effets induits en fautant des structures plus complexes dans le cœur du processeur.

Pour chaque programme de caractérisation, une simulation non fautée est d'abord réalisée pour collecter les résultats de référence. Puis, les injections sont réalisées pour chaque bascule cible. Les résultats des simulations fautées sont ensuite comparés aux résultats de référence afin de circonscrire les différences provoquées par l'injection (différences dans le banc de registres et la mémoire, pour chacun des deux points d'observation). Ces différences sont ensuite écrites dans une base de données que nous appellerons base de données RTL.

Il est à noter que même si l'observation se limite au banc de registres et à la mémoire, les effets induits dans d'autres structures pourraient également être observés ; ils ne sont cependant pas utilisés pour les comparaisons avec le niveau logiciel, par absence d'équivalent à ce niveau d'abstraction.

IV.3.2. Injection de faute RTL

Comme discuté plus tôt, l'injection de faute est réalisée au moment où l'instruction cible du programme de caractérisation est exécutée. Comme au niveau RTL les instructions ne sont pas exécutées en un seul cycle d'horloge, la faute peut donc être injectée à des instants différents. Pour s'assurer que la faute impacte bien l'instruction cible et pas une instruction adjacente, chaque bascule du processeur est associée à un ou plusieurs étages du pipeline. Par exemple, toutes les bascules des registres ID/EX de la Figure 10 sont associées à l'étage EX car une faute dans une de ces bascules impacte l'instruction actuellement dans l'étage d'exécution. Ainsi, spécifier une bascule-cible fixe en même temps le moment précis où la faute doit être injectée.

Certains effets comme l'exécution spéculative ou les défauts de cache (provoqués par l'injection d'une faute par exemple) peuvent faire varier le temps que met une instruction à s'exécuter. Pour trouver à quel moment la faute doit être injectée (et à quels moments les effets doivent être observés), il n'est donc pas possible de donner un temps fixe pendant la simulation. Il faut au contraire se fier à certains signaux du processeur. Lorsque c'est possible, les injections et observations sont synchronisées en observant la propagation du signal contenant l'adresse de l'instruction. Quand ces signaux ne sont pas trouvés (par exemple quand la faute impacte directement ces signaux de synchronisation), les observations sont effectuées en considérant que chaque étage du pipeline dure seulement un cycle d'horloge. Cette stratégie d'injection et d'observation semble un bon compromis entre complexité et précision des résultats, le but étant évidemment d'observer au même moment du côté RTL et du côté logiciel.

Les effets mettant en jeu la prédiction de branchement ou le cache peuvent rendre les observations très complexes : une même faute dans un programme de caractérisation peut donner des effets différents suivant l'état du cache ou de la prédiction de branchement. Pour avoir une certaine homogénéité et une reproductibilité des résultats observés, nous décidons donc de fixer ces états. Pour cela, les programmes de caractérisation sont au préalable exécutés quatre fois sans injection de faute. Expérimentalement, nous pouvons voir que cela permet de faire en sorte que les branchements soient correctement prédits et que les données utiles au programme soient déjà chargées dans la mémoire cache. Les injections sont ensuite réalisées pendant une cinquième exécution. L'impact de la prédiction de branchement et du cache pourrait faire l'objet d'une étude

spécifique, mais ils ajoutent de nombreuses difficultés (quelles lignes de cache charger, à quel niveau, quel historique de branchement, etc), c'est pourquoi nous les évitons ici. Notons pour finir que des défauts de cache ou des mauvaises prédictions peuvent tout de même être provoquées par la faute injectée ; seul l'état initial du système est fixé.

Les différents points exposés dans cette section peuvent paraître superficiels, mais sont pourtant capitaux, car les négliger compromettrait les comparaisons avec les injections logicielles.

IV.3.3. Observation des résultats

L'effet des fautes injectées est observé dans deux structures du processeur : le banc de registre et la mémoire. A chacun des deux points d'observation, l'état du banc de registres est simplement extrait de la simulation RTL. Pour l'observation de la mémoire, le procédé est cependant différent, pour plusieurs raisons. Tout d'abord parce que la mémoire contient un volume conséquent de données, et ensuite parce qu'il est relativement difficile d'avoir une vue globale de l'état de la mémoire à un instant donné, à cause notamment de la subdivision en différents niveaux de cache ou des transactions mémoires qui peuvent prendre un temps variable. La méthode retenue a donc été de simplement surveiller les transactions d'écriture émises par le processeur. Comme l'effet de la faute est déduit d'une comparaison entre une simulation fautée et une simulation non fautée, il n'est en effet pas nécessaire de connaître l'état global de la mémoire.

Ainsi, pour ce qui est de la mémoire, l'observation instantanée est constituée de l'éventuelle transaction d'écriture demandée par l'instruction cible. L'observation de propagation est quant à elle constituée de l'ensemble des transactions d'écriture effectuées entre les deux points d'observation (inclusif). Pour chaque transaction, ce sont l'adresse et la donnée écrite qui sont observées. Enfin, les possibles annulations de transactions (qui peuvent apparaître suite à une mauvaise spéculation par exemple) sont prises en compte.

IV.3.4. Classification des résultats d'injection

Pour chaque injection, après la comparaison avec la simulation non fautée, les résultats peuvent être classifiés dans les quatre catégories suivantes :

- silencieux : aucun effet visible (dans le banc de registres et la mémoire) à aucun des deux points d'observation.
- exception : une exception a été levée par le processeur durant la simulation. Dans ce cas, nous considérons que la faute est détectée.
- inconnu : l'un des points d'observation n'a pas été trouvé (par exemple si la faute a provoqué un saut trop lointain).
- faute analysable : toutes les injections qui n'appartiennent pas aux catégories précédentes.

Ces catégories sont semblables aux catégories fréquemment rencontrés dans les travaux similaires comme [15] et [65]. Les fautes dites analysables sont celles qui nous intéressent ; ce sont celles qui seront ensuite comparées aux injections logicielles. Les fautes inconnues ne sont pas prises en compte car il n'a pas été possible de retrouver un point d'observation, ce qui ne permet donc pas d'effectuer une comparaison valable avec des injections logicielles. Enfin, les fautes silencieuses et les fautes qui déclenchent une exception ne nous intéressent pas car elles ne créent aucun effet visible, ou sont considérées comme détectées par le processeur.

IV.4. Injection de faute au niveau logiciel

Dans cette partie, nous discutons de la manière de procéder pour effectuer les injections logicielles nécessaires à notre approche. La contrainte première est de pouvoir effectuer les injections de manière similaire aux injections RTL : même instruction cible et observation des mêmes structures au même moment.

IV.4.1. Développement d'un outil de mutation de programme

Contrairement au niveau RTL, il n'existe pas de méthode triviale pour injecter des fautes au niveau logiciel. Il existe des outils pour cela, discutés dans le 0, mais ceux-ci sont souvent liés à des modèles de faute logiciels particuliers. Or, au vu des conclusions du 0, il n'est pas possible de résumer tous les comportements fautifs dans un seul et même modèle de faute logiciel. Pour pouvoir étudier la robustesse d'une application face à la diversité des comportements obtenus, il est nécessaire d'avoir un système de représentation qui soit suffisamment générique pour accepter des modèles très différents les uns des autres, tout en permettant de modéliser des fautes qui mettent en jeu des particularités de la microarchitecture d'un processeur. Pour cela, nous avons créé un outil qui a pour principe de muter une application suivant un modèle de faute donné en paramètre.

IV.4.1.1. Spécifications de l'outil de mutation

Plusieurs contraintes importantes devaient être prises en compte :

- Le fichier d'entrée devait être le binaire de l'application plutôt que le code source ; et ce pour deux raisons. La première est que les instructions contenues dans le binaire correspondent à ce qui est réellement exécuté par le processeur, ce qui est important dans notre approche qui se veut très proche de la microarchitecture. Comparer des résultats d'injection n'a de sens que si les deux niveaux d'abstraction exécutent exactement la même séquence d'instructions. La seconde raison est que la compilation peut modifier la structure du programme, ajoutant des imprécisions sur les conclusions que l'on peut tirer au niveau source. La problématique de savoir comment le compilateur impacte la sécurité est complémentaire à ce travail ; nous nous intéressons donc directement à la sécurité du programme compilé.
- Comme les modèles de fautes doivent représenter des informations non disponibles au niveau binaire (comme l'état de registres cachés), les mutants générés par l'outil devaient être écrits à un autre niveau de représentation. Nous avons choisi de le représenter en langage C (d'autres auraient pu convenir) d'une part pour la simplicité de ce langage et d'autre part parce que nombre d'outils matures existent pour analyser un programme à ce niveau. Nous étudierons notamment dans le dernier chapitre de ce manuscrit l'utilisation d'outils d'analyse statique.
- Enfin, les modèles de fautes devaient pouvoir être facilement spécifiés ou modifiés pour tenir compte de la grande variété des effets possibles.

Les entrées et sorties de cet outil sont schématisées sur la Figure 33.

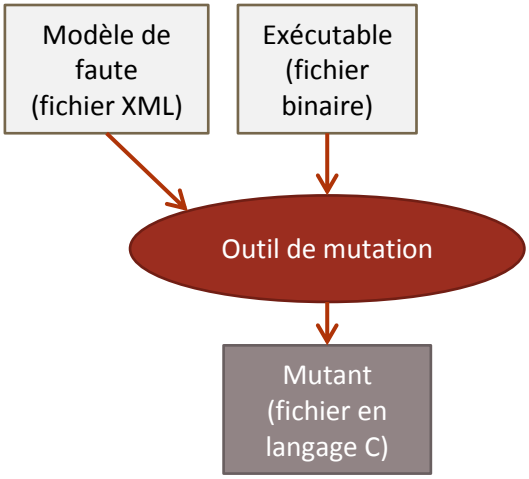

FIGURE 33 : ENTREES ET SORTIES DE L'OUTIL DE MUTATION

IV.4.1.2. Fonctionnement de l'outil de mutation

Mutation sans modèle de faute

La première fonction de l'outil de mutation est de traduire le code assembleur de l'application vers une représentation en langage C. Ce processus peut faire penser à un processus de décompilation. Ce n'est cependant pas le cas : le but n'est pas ici de retrouver le code source de l'application mais seulement de représenter le comportement du code assembleur au niveau C. Le programme résultant, même s'il est écrit en langage C, aura en réalité la même structure que le programme assembleur. Il y a une correspondance un-pour-un entre chaque instruction du binaire et son équivalent au niveau C.

Afin de faciliter la spécification des modèles de faute, l'exécution des instructions assembleur est divisée en trois phases, qui correspondent approximativement aux étages de décodage, d'exécution et d'écriture des résultats d'un pipeline classique de processeur. Ce choix de trois phases est un compromis qui permet de donner au développeur plus d'expressivité dans la modélisation des fautes, avec une complexité raisonnable. Un exemple de cette répartition en trois phases est montré dans la Figure 34, pour une instruction « ADDI x15 = x0 + 85 ».

FIGURE 34 : EXEMPLE DE REPRESENTATION EN C DE L'INSTRUCTION ADDI X15 = X0 + 85

Mutation avec un modèle de faute

Le but d'une injection de faute est de perturber l'exécution correcte d'une instruction. Dans le mutant généré, cette perturbation est modélisée en insérant des lignes de code entre les phases de décodage et d'exécution, ou entre les phases d'exécution et d'écriture des résultats. Nous voyons ici l'intérêt d'avoir représenté les instructions en trois phases : les modèles de faute permettent d'impacter l'exécution d'une instruction à différents endroits. Pour certains modèles, la faute a un impact avant la phase d'exécution (par exemple modifier la valeur d'un argument) alors que d'autres ont un impact plus tard (par exemple inversion du résultat d'un test conditionnel).

Les lignes de code à insérer pour les modèles de faute sont spécifiées dans un fichier XML séparé. Un exemple de modèle est montré en Figure 35. Le rôle de ce modèle est de remplacer le premier argument d'une instruction par le résultat de l'instruction précédemment exécutée. Cela représente donc une faute de forwarding. La spécification du modèle commence par la déclaration de variables nécessaires au modèle, dans la section **globals** sur la Figure 35 ; ici une variable *fwd* est déclarée pour représenter le rôle d'un registre caché qui stocke le résultat d'une instruction. Pour ce modèle, il y a deux actions à spécifier : la sauvegarde du résultat de la dernière instruction exécutée et le remplacement d'un argument par ce résultat. Deux distinctions doivent être opérées pour ces actions. La première action (sauvegarde du résultat) doit être appliquée à toutes les instructions exécutées alors que la seconde (remplacement d'un argument) ne doit être ajoutée qu'à l'instruction visée par l'injection. Pour distinguer ces deux cas, deux types de sections sont introduites : les sections de type **gold** sont insérées pour toutes les instructions ; les sections de type **fault** uniquement pour les instructions cibles de l'injection. La seconde distinction à effectuer est sur l'endroit où les lignes doivent être insérées. Celles-ci peuvent être insérées entre les phases de décodage et d'exécution de l'instruction, ce sont les sections de type **ini** ; ou entre les phases d'exécution et d'écriture des résultats, ce sont les sections de type **end**. Ces différents types de sections sont combinés, résultant en quatre sections : **gold_ini**, **gold_end**, **fault_ini** et **fault_end**. Pour le modèle en question, la sauvegarde du résultat d'une instruction doit être effectuée après la phase d'exécution et pour toutes les instructions du programme. Ainsi, le stockage du résultat *res* dans la variable *fwd* déclarée préalablement est effectué dans une section **gold_end**. Le remplacement du premier argument *arg1* d'une instruction doit être effectué avant la phase d'exécution et seulement pour l'instruction cible, d'où l'utilisation d'une section **fault_ini**.

```
<model name="exemple">
    <globals>     long fwd = 0;  </globals>
    <gold_end>  fwd = res;       </gold_end>
    <fault_ini>   if(injection_time==count) arg1=fwd; </fault_ini>
</model>
```

FIGURE 35 : EXEMPLE DE REPRESENTATION D'UN MODELE DE FAUTE

Pour éviter d'avoir à muter un programme trop souvent, le résultat du processus de mutation est en réalité un mutant d'ordre supérieur (également appelé méta-mutant) dans le sens où il contient des mutations dans chaque instruction cible où le modèle de faute en question est applicable ; et que c'est lors de l'exécution que la mutation à utiliser est choisie grâce au paramètre *injection_time.* Une variable *count* est incrémentée à chaque fois qu'une instruction cible est exécutée et quand cette variable atteint la valeur spécifiée dans *injection_time*, l'injection est effectuée.

Au final, la mutation de l'instruction de la Figure 34 avec le modèle de la figure Figure 35 est représentée en Figure 36.

```
l06ac: // ADDI x15, x0, 85
        count++;
        arg1 = reg[0];  arg2 = 85;  // Decode
        if(injection_time==count) arg1=fwd;
        res = arg1 + arg2;          // Execute
        fwd=res;
        reg[15]=res;                // Write-Back
```

FIGURE 36: EXEMPLE DE MUTATION DE L'INSTRUCTION ADDI X15, X0, 85

IV.4.1.3. Limitations de l'outil de mutation

L'outil de mutation possède quelques limitations qu'il est important de garder en tête. Tout d'abord, dans sa version actuelle, il ne supporte que les instructions du jeu d'instructions de base (instructions sur les entiers) ainsi que celles de l'extension de multiplication/division. En d'autres termes, il ne peut être utilisé que pour une architecture RISC-V IM de 64 bits. Les instructions de gestion des registres de statut et de contrôle (CSR) ne sont également pas supportées.

Une des difficultés lorsque l'on passe d'une représentation assembleur à une représentation en langage C réside dans la gestion des sauts dont la cible n'est pas connue à l'avance. Dans le jeu d'instruction RISC-V, il existe l'instruction JALR qui permet de sauter à une adresse spécifiée dans un registre assembleur. Cette instruction n'a pas d'équivalent en C ; seuls les sauts statiques sont possibles par l'intermédiaire de l'instruction *goto*. La gestion de l'instruction JALR est donc limitée ; seuls les sauts dont la cible peut être facilement déterminée, comme les retours de fonctions, sont possibles.

Cette difficulté à gérer dynamiquement le flot de contrôle au niveau C a également une répercussion sur les modèles de faute. En effet, il rend difficile la spécification de modèles qui influent directement sur le flot de contrôle. Les modèles simples comme le saut d'instruction peuvent être spécifiés avec un peu d'effort, mais des modèles plus complexes ne sont pas traités.

IV.4.1.4. Validation de l'outil de mutation

Afin de vérifier le fonctionnement de l'outil de mutation, une phase de validation a été effectuée. Le but de cette validation était de s'assurer de l'exécution correcte du mutant en l'absence de fautes.

Pour effectuer cette validation, les vecteurs de test fournis avec la suite d'outils RISC-V ont été utilisés. Ceux-ci sont, en principe, utilisés pour vérifier, par exemple, le fonctionnement d'une implémentation de processeur. Ces vecteurs de tests consistent à vérifier que chaque instruction prise séparément donne le résultat attendu, dans différents contextes d'exécution, avec des données différentes et en testant des cas limites comme les débordements. Chaque instruction est ainsi vérifiée avec entre une dizaine et une quarantaine de tests, en fonction du type d'instruction.

Quant à elle, la validation de l'outil de mutation en présence de faute revient directement à valider les modèles de faute logiciels, en comparant les résultats à ceux d'injections au niveau RTL ; ce qui est l'objet du reste de ce chapitre.

IV.4.2. Méthode d'injection logicielle

Nous disposons maintenant d'un moyen d'injecter des fautes au niveau logiciel. Le bloc « injections logicielles » du diagramme de notre approche, vu en début de chapitre (Figure 29), peut donc être expliqué. Le processus d'injection logicielle, illustré en Figure 37, est similaire à l'injection RTL, avec cependant une étape préliminaire en plus ; celle de la mutation du programme de caractérisation.

Chaque programme de caractérisation est envoyé à l'outil de mutation avec un modèle de faute logiciel, afin de générer un mutant. Ce mutant est compilé puis exécuté plusieurs fois pour obtenir les résultats d'une exécution non fautée et les résultats d'exécutions fautées. L'exécution non fautée pourrait également être réalisée sans générer de mutant, mais il est plus simple de le faire de cette manière car le processus est quasiment identique (l'injection de faute n'est tout simplement pas activée). Ces résultats sont comparés puis stockés dans une base de données appelée base de données logicielle.

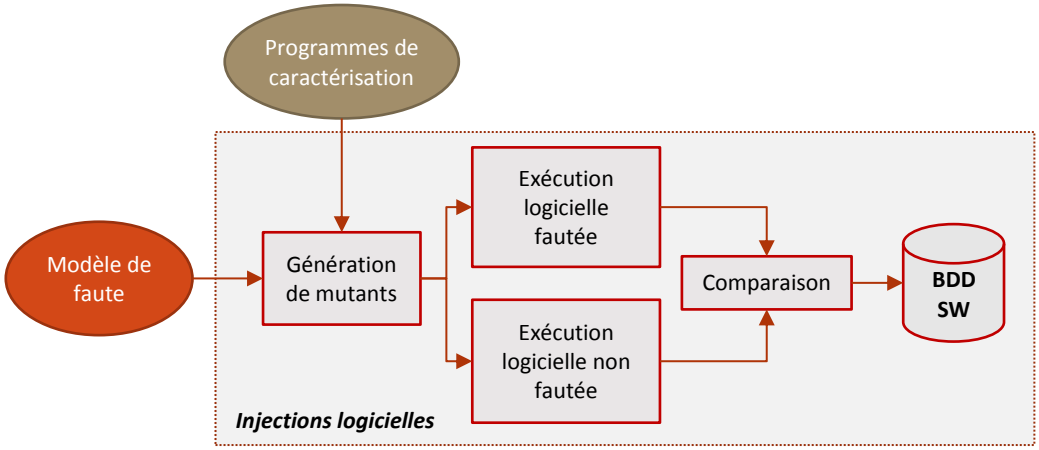

FIGURE 37 : VUE D'ENSEMBLE DE L'INJECTION LOGICIELLE. LES PROGRAMMES DE CARACTERISATION SONT UTILISES DANS DES EXECUTIONS FAUTEES ET NON FAUTEE. LES RESULTATS SONT COMPARES ET STOCKES DANS UNE BASE DE DONNEES

Tout comme pour les injections RTL, les résultats peuvent être répartis en quatre catégories : faute silencieuses (aucuns effets aux deux points d'observation), exception (problème survenu pendant l'exécution du mutant), inconnu (point d'observation non trouvé) et enfin fautes analysables (tous les autres cas).

Les mêmes règles d'observation que pour le côté RTL sont utilisées ; et en particulier, le banc de registres est observé aux deux points d'observation, et pour la mémoire ce sont les transactions d'écriture qui sont enregistrées.

IV.5. Analyses multi-niveaux et calcul de métriques

Une fois les campagnes d'injection RTL et logicielle réalisées, vient la troisième et dernière partie de notre approche : la comparaison et l'analyse des résultats. Dans cette partie, nous présentons tout d'abord un formalisme mathématique pour décrire précisément les conditions et résultats des injections présentées dans les sections précédentes. Ce formalisme est ensuite utilisé pour développer différents types d'analyses utiles pour les processus d'abstraction et de concrétisation évoqués dans l'introduction de ce chapitre.

IV.5.1. Formalisme mathématique

IV.5.1.1. Modélisation des paramètres d'injection

Nous définissons tout d'abord l'ensemble **C** des *programmes de caractérisation*. Ces programmes incluent une instruction cible et une instruction d'observation, comme définies précédemment.

Nous définissons ensuite l'ensemble **M** des *méthodes d'injection*. La *méthode d'injection* diffère en fonction du type d'injection utilisé. Pour une injection RTL, la *méthode d'injection* est l'ensemble des bascules qui sont fautées pendant l'exécution de l'instruction cible. Pour une injection logicielle, la *méthode d'injection* est le nom du modèle de faute utilisé.

L'ensemble **E** est l'ensemble des *effets* d'une faute à un point donné du programme. Un *effet* est défini comme les différences visibles (banc de registres et mémoire) qui existent entre une exécution fautée et une exécution non fautée du programme de caractérisation. Par exemple, si dans un programme de caractérisation donné, une exécution fautée finit dans le même état que l'exécution non fautée, excepté le registre 10 qui a la valeur 1 au lieu de 0 et le registre 12 qui a la valeur 42 ou

lieu de -42, alors l'*effet* $e \in \boldsymbol{E}$ est noté « R(10:0:1)(R12:-42:42) ». S'il existe aussi des différences dans la mémoire, alors l'ensemble des transactions d'écriture sont également notées.

L'ensemble $\boldsymbol{B} \equiv \boldsymbol{C} \times \boldsymbol{E} \times \boldsymbol{E}$ est l'ensemble des *comportements* provoqués par des injections de faute. Un *comportement* est constitué d'un programme de caractérisation (qui définit à quel moment injecter et observer), un effet instantané et un effet de propagation, conformément à la méthode d'injection présentée précédemment. Un *comportement* décrit donc tous les effets provoqués pendant une injection, indépendamment de la méthode d'injection (donc que ce soit au niveau RTL ou au niveau logiciel).

Enfin, un *résultat d'injection* est un élément de l'ensemble $\boldsymbol{R} \equiv \boldsymbol{B} \times \boldsymbol{M}$**.** Un *résultat d'injection* contient les mêmes informations qu'un *comportement*, avec en plus la *méthode d'injection* utilisée pour obtenir ce comportement. Ainsi, un élément $r \in \boldsymbol{R}$ décrit complètement les paramètres et effets d'une injection de faute. Le Tableau 4 résume toutes ces définitions.

Avec ces définitions, une *injection de faute* est simplement une fonction $\boldsymbol{C} \times \boldsymbol{M} \rightarrow \boldsymbol{R}$ qui lie un programme de caractérisation et une méthode d'injection à un résultat d'injection. Dans notre cas, ces fonctions correspondent aux processus d'injection décrits dans les parties IV.3 et IV.4, que nous appellerons f_{RTL} et f_{SW}, respectivement.

TABLEAU 4 : RESUME DES DEFINITIONS DU FORMALISME MATHEMATIQUE

Ensemble	Nom	Exemple	Signification
C	Programmes de caractérisation	$c =$ "prog1".	L'injection est réalisée pendant l'exécution du programme *prog1*, qui contient une instruction cible et une instruction où est effectuée l'observation des effets de propagation.
M	Méthodes d'injection	$m_1 =$ "$b1\ b2$" $m_2 =$ "$modèle1$"	Injection RTL dans les bascules *b1* et *b2*. Injection logicielle avec le modèle *modèle1*.
E	Effets d'injection	$e =$ "(R10:0:1)(R12:-42:42)"	A un certain point de l'exécution, le registre 10 devrait valoir 0, mais vaut 1; et le registre 12 devrait valoir -42 mais vaut 42.
B	Comportement d'injection	$b = (c, e_1, e_2)$	Injecter une faute dans le programme *c* a permis d'obtenir l'effet instantané e_1 et l'effet de propagation e_2.
R	Résultat d'injection	$r = (b, m)$	Le comportement b a été obtenu avec la méthode d'injection *m*.

Toutes ces définitions permettent de visualiser les campagnes d'injection sous forme d'ensembles représentés en Figure 38. Les programmes de caractérisation définis dans l'ensemble C_0 sont utilisés pour les deux côtés de l'approche. Il faut cependant séparer les méthodes d'injection. Nous distinguerons donc M_{RTL}, l'ensemble des méthodes d'injection du côté RTL (i.e. les bascules à fauter) et M_{SW}, l'ensemble des méthodes d'injection logicielles (i.e. les modèles de faute logiciels). Injecter des fautes dans les programmes de C_0 avec les méthodes de M_{RTL} permet de créer l'ensemble de résultats d'injection R_{RTL}. De même, injecter des fautes dans les programmes de C_0 avec les méthodes M_{SW} permet de créer l'ensemble R_{SW}.

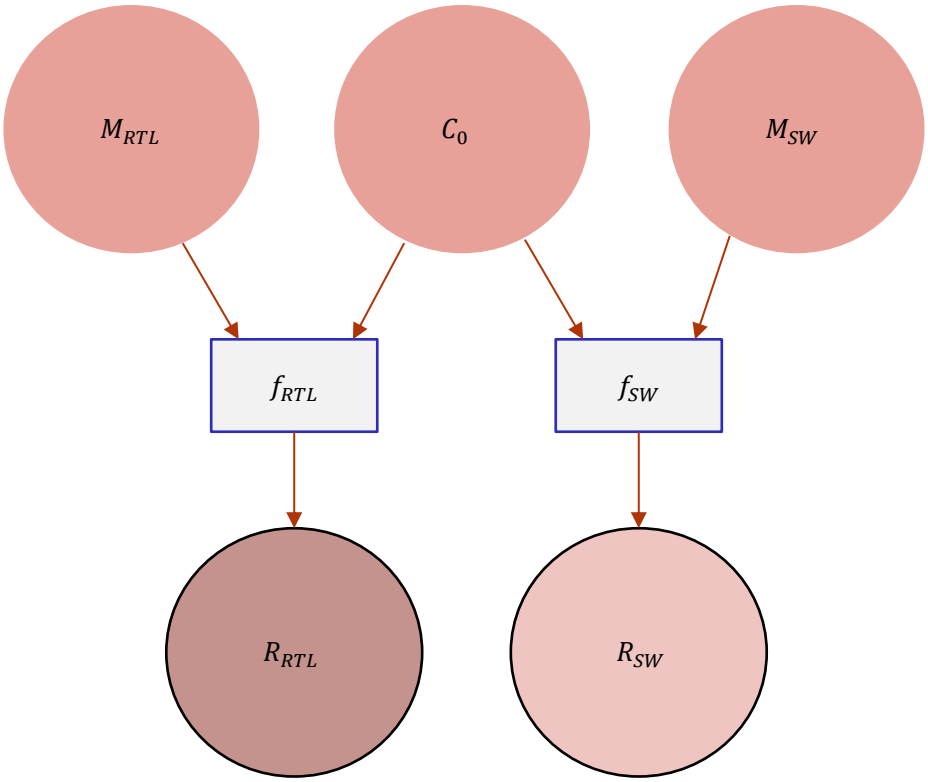

FIGURE 38 : DIAGRAMME DES CAMPAGNES D'INJECTION DE FAUTES. LES INJECTIONS AU NIVEAU RTL, f_{RTL}, PRENNENT EN ENTREE LES BASCULES A FAUTER M_{RTL} ET LES PROGRAMMES DE CARACTERISATION DE C_0, ET GENERENT LES RESULTATS D'INJECTION R_{RTL}. DE MEME, LES INJECTIONS AU NIVEAU LOGICIEL f_{SW} PRENNENT EN ENTREE LES MEMES PROGRAMMES DE CARACTERISATION DE C_0, AINSI QUE LES MODELES DE FAUTE DEFINIS DANS M_{SW} POUR PRODUIRE DES RESULTATS D'INJECTIONS R_{SW}.

La fonction f_{RTL} représente les injections au niveau RTL, donc crée des résultats d'injection qui sont considérés comme la référence. f_{SW} représente les injections logicielles réalisées au moyen de l'outil de mutation. Le but de l'approche est que les résultats produits par f_{SW} ressemblent à ceux de f_{RTL}, avec un ensemble de départ différent (modèles de faute logiciels au lieu de bascules au niveau matériel).

IV.5.1.2. Comparaison de résultats d'injection

La comparaison entre les résultats obtenus côté RTL et côté logiciel n'a de sens que pour les fautes analysables telles que définies dans les sections IV.3.4 et IV.4. Parmi les résultats d'injection R_{RTL}, nous distinguons donc le sous-ensemble R_{RTL}^A de fautes analysables et nous définissons de même le sous-ensemble R_{SW}^A de R_{SW}.

Comme les *résultats d'injection* du côté RTL et du côté logiciel ont des *méthodes d'injection* différentes (bascules dans le premier cas, modèles de faute logiciels dans le second), ces résultats ne peuvent se recouper. Pour pouvoir comparer les effets des fautes, il est alors nécessaire de définir la projection suivante :

$$\boldsymbol{\pi} : R \rightarrow B$$
$$\boldsymbol{\pi}(b, m) = b$$

Cette opération prend un résultat d'injection et le projette vers son comportement correspondant. Plusieurs résultats d'injection peuvent aboutir à un même comportement ; dans ce cas, le nombre de pré-images correspond à la *surface d'attaque* associée à ce comportement, c'est-à-dire le nombre de bascules qui, lorsqu'elles sont ciblées produisent le comportement en question.

Avec la projection **π,** un résultat d'injection RTL correspond à un résultat d'injection logiciel si et seulement si l'injection est effectuée au même point de l'exécution du programme de caractérisation et si les effets instantanés et effets de propagation sont identiques pour les deux modes d'injection. Cette comparaison est relativement stricte dans le sens où trois paramètres doivent être identiques ; une projection moins restrictive sera étudiée ultérieurement.

Comme la projection **π** se sépare de la dimension *méthode d'injection*, il devient possible d'étudier les recoupements entre les comportements réels générés au niveau RTL et les comportements prédits au niveau logiciel. Ce recoupement est illustré en Figure 39 ; c'est la zone de recoupement des cercles $\boldsymbol{\pi}(R^A_{RTL})$ et $\boldsymbol{\pi}(R^A_{SW})$.

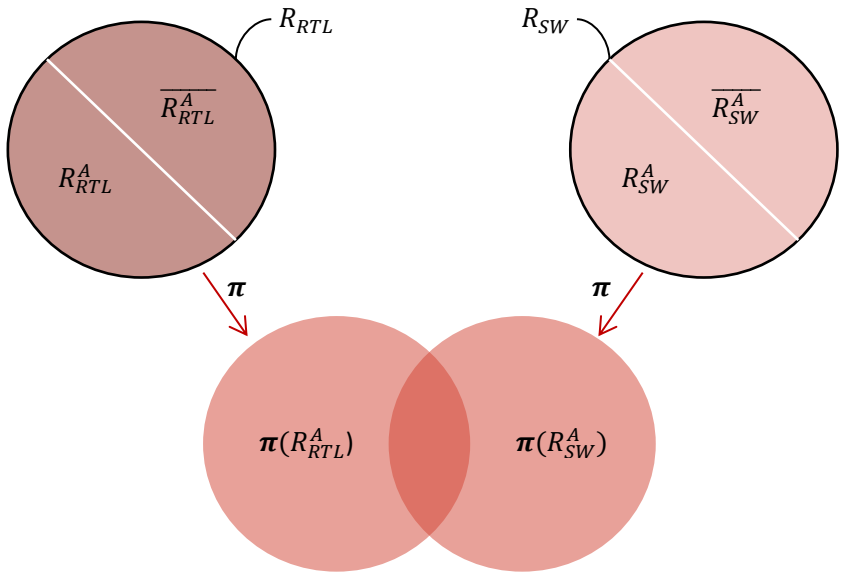

FIGURE 39 : REPRESENTATION DU RECOUPEMENT DES COMPORTEMENTS RTL ET LOGICIELS. LE CERCLE EN HAUT A GAUCHE REPRESENTE LES RESULTATS D'INJECTION RTL ET CELUI EN HAUT A DROITE, LES RESULTATS D'INJECTION LOGICIELLE. CES DEUX CERCLES PEUVENT ETRE DIVISES EN FAUTES ANALYSABLES ET NON-ANALYSABLES. LES PARTIES ANALYSABLES PEUVENT ENSUITE ETRE PROJETEES VERS L'ESPACE DES COMPORTEMENTS POUR ETUDIER LEUR RECOUPEMENT.

Mathématiquement, pour définir si un comportement logiciel correspond à un comportement RTL (ou vice-versa), nous utiliserons la fonction caractéristique suivante, avec D un ensemble de comportements :

$$\mathbf{1}_D : B \rightarrow \{0,1\}$$
$$\mathbf{1}_D(b) = \begin{cases} 1 \ si \ b \in D \\ 0 \ sinon \end{cases}$$

IV.5.2. Analyses du processus d'abstraction

Grâce au formalisme mathématique tout juste présenté, il est possible d'étudier tout d'abord le processus d'abstraction, c'est-à-dire le passage de fautes au niveau RTL vers des fautes au niveau logiciel. Nous commencerons par présenter deux types de métriques pour évaluer la pertinence de modèles de faute logiciels, puis nous présenterons deux méthodes pour aider à la modélisation logicielle de fautes RTL.

IV.5.2.1. Métrique de couverture

Commençons par définir des métriques de couverture, pour chercher à évaluer le nombre de résultats d'injections RTL correctement prédits par les modèles de faute logiciels. Tout d'abord, nous dirons que les modèles de faute définis dans M_{SW} **couvrent** un résultat d'injection $r \in R^A_{RTL}$, si et seulement si :

$$\mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}\big(\boldsymbol{\pi}(r)\big) = 1$$

En d'autres termes, les modèles de faute logiciels couvrent un résultat d'injection RTL si et seulement si la projection du résultat d'injection arrive dans la partie où se situe le chevauchement dans la Figure 39. En sommant sur tous les éléments de R^A_{RTL} et en divisant par le nombres de fautes représentées dans R_{RTL}, nous pouvons ainsi définir la *couverture globale* (GC) des modèles de faute logiciels :

$$GC = \frac{\sum_{r \in R^A_{RTL}} \mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}(\boldsymbol{\pi}(r))}{|R_{RTL}|}$$

La *couverture globale* donne le pourcentage de fautes RTL prédites par les modèles de faute logiciels, relativement à toutes les injections. Le pourcentage peut donc être comparé directement au pourcentage de fautes RTL analysables. Pour avoir le pourcentage relatif seulement aux fautes analysables, nous définissons également la *couverture des fautes* (FC). Seul le dénominateur change :

$$FC = \frac{\sum_{r \in R^A_{RTL}} \mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}(\boldsymbol{\pi}(r))}{\left|R^A_{RTL}\right|}$$

La *couverture globale* et la *couverture des fautes* intègrent la notion de surface d'attaque : plusieurs injections (dans des bascules différentes) résultant en un même comportement seront comptées plusieurs fois. Mais il peut également être intéressant de ne les comptabiliser qu'une fois et se focaliser ainsi sur les comportements plutôt que les résultats d'injection. Nous définissons ainsi la *couverture comportementale* (BC):

$$BC = \frac{\sum_{b \in \boldsymbol{\pi}(R^A_{RTL})} \mathbf{1}_{\boldsymbol{\pi}(R^A_{SW})}(b)}{\left|\boldsymbol{\pi}(R^A_{RTL})\right|}$$

Techniquement, les analyses de couverture se font en croisant les données de la base de données RTL avec la base de données logicielle.

IV.5.2.2. Métrique de justesse

Le passage d'un niveau d'abstraction donné à un niveau d'abstraction plus élevé s'accompagne d'une perte de précision. En l'occurrence, les modèles de faute logiciels ne sont pas parfaits : il existe des approximations, notamment du fait que l'on n'observe que les effets visibles des fautes. Un résultat d'injection obtenu par injection logicielle n'est pas forcément réalisable en simulation RTL. Il est alors

intéressant d'évaluer la **justesse** des modèles de faute logiciels, c'est-à-dire leur capacité à prédire des effets réels et non des effets qui n'existent pas en réalité.

Cette étude de justesse est en quelque sorte l'inverse de l'étude de couverture. Dans l'étude de couverture, nous avons regardé pour chaque faute RTL s'il existait une faute logicielle qui donnait les mêmes résultats, ce qui signifiait qu'au moins un modèle logiciel permettait de prédire la faute. Pour l'étude de justesse, nous regardons pour chaque faute logicielle s'il existe une faute RTL qui donne les mêmes résultats, ce qui signifie dans ce cas que la faute prédite au niveau logiciel correspond à un comportement réel du processeur.

La justesse d'un modèle ou d'un ensemble de modèles peut tout d'abord être représentée sous forme d'un diagramme comme sur la Figure 40. Dans ce diagramme, les différents résultats des injections de faute sont groupés en cinq catégories :

- Fautes correctes : fautes logicielles qui prédisent correctement un effet obtenu dans une injection RTL.
- Fautes correctes mais redondantes : cas redondants de la catégorie précédente (car plusieurs modèles peuvent aboutir à un même comportement).
- Fautes non analysables : cas pour lesquels aucune conclusion ne peut être tirée quant à la justesse (le modèle logiciel ne crée aucun effet visible, ou l'exécution du mutant a rencontré un problème).
- Fautes incorrectes : fautes logicielles qui prédisent un comportement non observé au niveau RTL.
- Fautes incorrectes mais redondantes : cas redondants de la catégorie précédente.

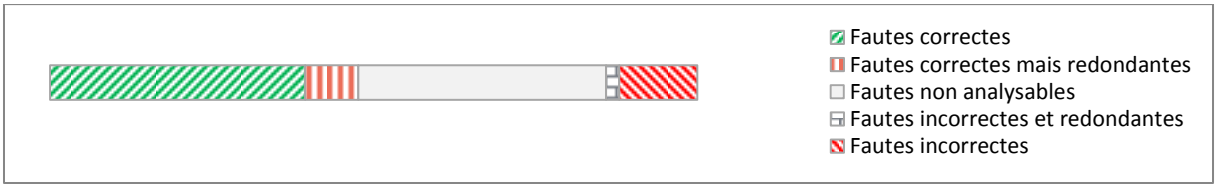

FIGURE 40 : DIAGRAMME DE JUSTESSE GLOBAL D'UN ENSEMBLE DE MODELES DE FAUTE LOGICIELS.

Ce diagramme permet donc de donner rapidement une idée de la justesse d'un ensemble de modèles logiciels et du niveau de redondance entre les modèles.

Les fautes redondantes n'apportent pas de nouvelles informations dans une analyse de sécurité logicielle (contrairement au niveau RTL où obtenir le même comportement en fautant des bascules différentes est lié au concept de surface d'attaque) et les fautes non analysables (principalement les fautes qui n'ont aucun effet visible dans un programme de caractérisation donné) n'apportent pas d'information non plus. En ignorant ces cas, il est possible de construire le diagramme en Figure 41.

FIGURE 41 : DIAGRAMME DE JUSTESSE REDUIT D'UN ENSEMBLE DE MODELES DE FAUTE LOGICIELS.

Numériquement, la *justesse* (J) d'un modèle ou ensemble de modèles, qui correspond au rapport visible en Figure 41, peut alors s'exprimer de la façon suivante :

$$J = \frac{\sum_{b \in \boldsymbol{\pi}(R_{SW}^{A})} \mathbf{1}_{\boldsymbol{\pi}(R_{RTL}^{A})}(b)}{\left|\boldsymbol{\pi}(R_{SW}^{A})\right|}$$

Cette formule correspond à la proportion de fautes correctes comme sur la Figure 41. Elle a exactement la même structure que la formule de la couverture comportementale, mais les rôles des résultats d'injection RTL et logiciels sont inversés. Un ensemble de modèles ayant une justesse de 75% signifie que dans trois cas sur quatre, les effets prédits par les modèles correspondent à des effets réels alors que dans un cas sur quatre, les effets ne correspondent pas à la réalité mais sont dus aux approximations des modèles logiciels.

IV.5.2.3. Recherche de fautes non couvertes

Si la couverture d'un ensemble de modèles de faute logiciels n'est pas satisfaisante, il est nécessaire d'améliorer ces modèles ou d'en ajouter. Dans ce cas, il peut être intéressant de souligner les comportements RTL non encore couverts qui se manifestent le plus souvent. Cela peut donner une indication sur la voie à suivre pour améliorer le plus efficacement la couverture.

Plus formellement, le but est de trouver $b \in \boldsymbol{\pi}(R_{RTL}^{A})$ tel que $b \notin \boldsymbol{\pi}(R_{SW}^{A})$ et que b a le plus grand nombre de pré-images de $\boldsymbol{\pi}$ dans R_{RTL}^{A}.

Dans l'approche adoptée, ce processus est relativement simple. Il suffit d'ordonner les comportements de la base de données RTL selon leur fréquence d'apparition (nombre de bascules qui peuvent provoquer ce comportement) et d'éliminer ceux qui sont également présents dans la base de données logicielle (c'est-à-dire ceux qui sont déjà prédits par les modèles logiciels).

IV.5.2.4. Sélection d'un ensemble de modèles de faute logiciels

Une fois toute l'approche automatisée, il est facile d'évaluer rapidement la pertinence de divers modèles de faute logiciels. Ajouter de nouveaux modèles nécessite simplement de lancer de nouvelles injections logicielles et de recalculer la couverture et la justesse. Le danger est alors de se retrouver avec trop de modèles logiciels de telle manière qu'une analyse de sécurité au niveau logiciel devienne trop coûteuse. L'intérêt principal d'une analyse au niveau logiciel étant de gagner du temps par rapport à une analyse au niveau matériel, considérer trop de modèles logiciels pourrait se révéler contre-productif. Dans ce cas, il peut alors être intéressant de sélectionner parmi l'ensemble de modèles ceux qui représentent le mieux les comportements du processeur en question.

Le critère principal pour évaluer la pertinence d'un modèle de faute logiciel étant sa couverture, nous voulons alors sélectionner un sous-ensemble de M_{SW} qui maximise la *couverture globale GC*. Cette question est équivalente au problème de la couverture maximale, un problème classique en informatique. Ce problème fait partie de la classe de problèmes NP-difficile, ce qui signifie, sommairement, qu'il n'existe pas de méthode pour résoudre ce problème dans un temps raisonnable. Un algorithme glouton peut cependant aboutir à une bonne approximation du résultat [27]. L'algorithme glouton consiste simplement à sélectionner les modèles logiciels un par un, en choisissant à chaque fois celui qui couvre le plus de fautes non déjà couvertes.

L'algorithme de sélection de modèle pourrait également être utilisé pour optimiser d'autres paramètres comme la *couverture comportementale* ou la *justesse*.

IV.5.3. Analyses du processus de concrétisation

Dans les sections précédentes, l'étude était focalisée sur le processus d'abstraction de fautes du niveau RTL vers le niveau logiciel. Mais comme évoqué dans l'introduction de ce chapitre, il est également possible de raisonner sur le processus inverse, le processus de concrétisation.

IV.5.3.1. Profils de modèles de faute logiciels

La première analyse qui peut être faite concerne la représentation du *profil* d'un modèle de faute logiciel. Le but d'un *profil* est de représenter, sous forme d'un diagramme, les bascules du processeur qui, lorsqu'elles sont cibles d'une injection, sont les plus à-même de recréer l'effet d'un modèle de faute logiciel donné.

La représentation de ce profil est réalisée de la façon suivante. Pour chaque bascule $m \in M_{RTL}$, nous comptons le nombre de programmes de caractérisation pour lesquels le comportement obtenu après injection est identique à celui créé par un modèle logiciel donné ; puis nous trions ces bascules pour identifier celles qui correspondent le plus fréquemment au modèle logiciel.

Nous obtenons ainsi un diagramme semblable à la Figure 42. Sur cette figure, nous voyons que pour reproduire au niveau RTL les mêmes comportements que le modèle logiciel associé, les bascules **bascule_1** et **bascule_2** sont les meilleurs candidats. Injecter dans les autres bascules peut parfois résulter dans les mêmes comportements, mais cela est moins fréquent.

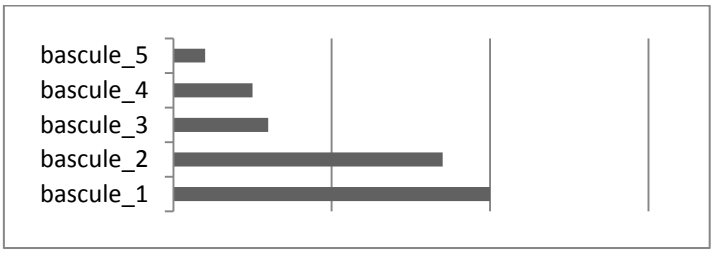

FIGURE 42 : EXEMPLE DE PROFIL D'UN MODELE LOGICIEL LAMBDA

L'utilité de ces *profils* est de pouvoir vérifier rapidement si un comportement observé au niveau logiciel est effectivement réalisable en réalité, en effectuant des injections RTL ciblées. Cela est particulièrement intéressant dans le cas où le modèle logiciel a une faible justesse. Si dans un système, un vecteur d'attaque est trouvé avec un modèle logiciel peu fiable, faire quelques injections RTL pour vérifier si cette attaque est effectivement possible peut aider dans la décision de protéger ou non.

Ces profils pourraient également servir à construire une cartographie du processeur, montrant spatialement où sont placées les bascules susceptibles de reproduire le comportement d'un modèle logiciel. Cela peut être utile pour guider des injections expérimentales au laser par exemple.

IV.5.3.2. Identification de bascules à protéger

Le processus de concrétisation peut également servir à aider à choisir quelles parties du processeur devraient être protégées au niveau matériel.

Pour cela, nous pouvons tout d'abord trier les bascules de M_{RTL} en fonction du nombre de comportements qu'elles créent en leur injectant une faute pendant l'exécution des programmes de caractérisation de C_0. Nous connaissons ainsi les bascules qui ont le plus souvent un effet visible sur l'exécution d'un programme et donc celles qu'il faudrait protéger en priorité. Ce tri peut se faire en considérant uniquement les injections RTL, mais nous pouvons également aller plus loin en éliminant

les cas qui sont correctement prédits par les modèles de faute logiciels. Les attaques connues au niveau logiciel peuvent potentiellement être contrées à ce niveau ; ce sont les comportements non prédits qui sont plus problématiques. Ainsi, savoir quelles bascules ont des comportements bien définis au niveau logiciel peut aider à prendre la décision de quelles bascules protéger au niveau matériel.

Sur la Figure 43, **bascule_1** crée le plus de comportements fautifs, mais plus de la moitié de ces comportements sont correctement prédits au niveau logiciel. Au contraire, **bascule_2**, qui crée également beaucoup de comportements fautifs, est mal modélisée au niveau logiciel ; une contremesure matérielle sur cette bascule peut donc avoir plus de sens que sur la précédente.

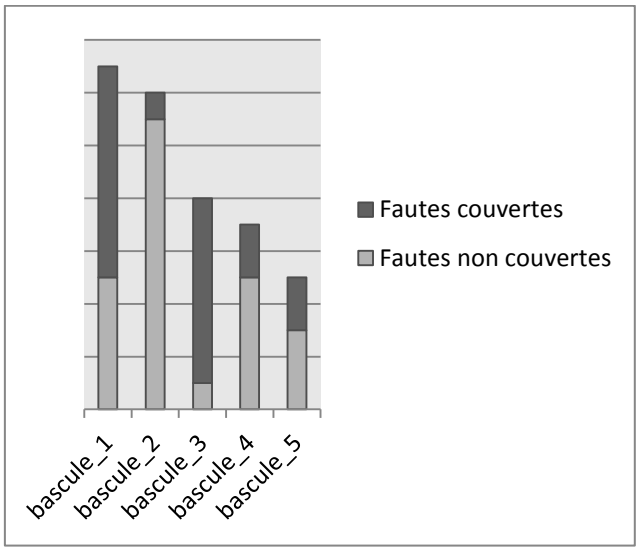

FIGURE 43 : EXEMPLE DE DIAGRAMME POUR IDENTIFIER LES BASCULES A PROTEGER

Ici ce sont donc des informations du côté logiciel qui permettent de prendre de meilleures décisions au niveau matériel. La connaissance d'informations au niveau logiciel modifie l'ordre des priorités au niveau matériel. Nous sommes bien dans un processus de concrétisation.

IV.5.4. Conclusion sur les analyses

Dans cette section, nous avons défini un formalisme mathématique pour décrire les campagnes d'injection de fautes et nous avons ensuite développé divers types d'analyse que l'on peut classer en deux catégories. D'une part les analyses d'abstraction :

- Calcul de métriques de couverture pour évaluer la proportion d'injection RTL prédites par les modèles logiciels
- Calcul d'une métrique de justesse pour évaluer la proportion de prédictions correctes d'un modèle logiciel donné
- Recherche des cas non couverts les plus fréquents
- Sélection des meilleurs modèles logiciels

Et d'autre part, les analyses de concrétisation :

- Représentation des profils des modèles pour savoir à quelles structures du processeur correspondent les modèles logiciels

- Identification des bascules à protéger en priorité, en tenant compte de celles correctement modélisées au niveau logiciel

Notons que les analyses présentées sous l'aspect concrétisation peuvent complémenter celles de l'aspect abstraction : la *justesse* des modèles peut être complémentée par des injections RTL guidées par les *profils* ; et l'*identification de bascules à protéger* permet de distinguer les bascules dont les comportements fautifs sont peu *couverts* par les modèles logiciels.

Ces différents types d'analyse ont été définis de manière théorique ; il est maintenant temps de les appliquer sur un cas pratique.

IV.6. Résultats sur un cas pratique

Dans cette partie, les différentes métriques et analyses sont appliquées à un cas concret, pour illustrer l'intérêt de l'approche et tirer des conclusions sur la modélisation de fautes dans un processeur d'architecture RISC-V.

Les différentes entrées de la méthode sont tout d'abord explicitées, avant de passer à l'application des métriques et analyses.

IV.6.1. Paramètres d'entrée

Trois ensembles d'entrée doivent être définis, comme montrés dans la Figure 29 et la Figure 38. Nous définirons donc dans cette section l'ensemble des programmes de caractérisation (ensemble C_0), puis l'ensemble des bascules cibles des injections (ensemble M_{RTL}), et enfin l'ensemble des modèles de faute logiciels (ensemble M_{SW}).

IV.6.1.1. Programmes de caractérisation : l'ensemble C_0

Premier ensemble : contextes assembleur

Les contextes assembleur sont de petits segments de code écrits au niveau assembleur. Comme défini dans la section IV.2.2, ces contextes sont construits à partir d'un prologue, une cible et un épilogue.

Ici, nous nous concentrons sur les instructions du jeu d'instruction de base de RISC-V (architecture RV64I). Comme l'étude préliminaire du chapitre précédent a montré de nombreux effets provenant de la structure de forwarding du processeur, les contextes sont bâtis en particulier pour faire apparaître différentes situations de dépendances entre instructions consécutives.

Tous les prologues, cibles et épilogues utilisés dans cette étude pratique sont répertoriés dans les Tableau 5, Tableau 6 et Tableau 7, respectivement. La Figure 44 montre un exemple de contexte assembleur construit à partir de ces tableaux.

TABLEAU 5 : LISTE DE PROLOGUES

Nom du prologue	Instructions
Nofwd	ADDI a0, x0, 5; ADDI a1, x0, -15; NOP; NOP;
Fwdone	ADDI a0, x0, 5; ADDI a1, x0, -15; NOP;
Fullfwd	ADDI a0, x0, 5; ADDI a1, x0, -15;

TABLEAU 6 : LISTE DE CIBLES

Nom de la cible	Instructions
Add	ADD a2, a0, a1;
Add_r	ADD a2, a1, a0;
Addi	ADDI a2, a1, 37;
Lw	LW a2, 52(tp);
Sw	SW a1, 16(tp);
Bltz	BLTZ a1, label;
Jal	JAL label;

TABLEAU 7 : LISTE D'EPILOGUES

Nom de l'épilogue	Instructions
Nop3	NOP;NOP;NOP;
Fwdfst	ADDI a3, a2, 78; NOP;
Fwdsnd	NOP; ADDI a4, a2, 78; NOP;
Fwdboth	ADDI a3, a2, 78; ADDI a4, a2, 78; NOP;
Sb	SB a2, 16(tp); NOP;

```
ADDI a0, x0, 5;
ADDI a1, x0, -15;
ADD  a2, a0, a1;      // Injection
ADDI a3, a2, 78;
NOP;                  // Observation
```

FIGURE 44 : CONTEXTE ASSEMBLEUR FORME A PARTIR DU PROLOGUE "FULLFWD", LA CIBLE "ADD" ET L'EPILOGUE "FWDFST"

En construisant toutes les combinaisons des trois prologues avec les sept cibles et les cinq épilogues, nous obtenons un total de 105 contextes assembleur. D'autres prologues, cibles et épilogues pourraient être ajoutés afin d'étudier encore d'autres situations, mais pour ne pas biaiser les analyses qui suivent, il est nécessaire de conserver un certain équilibre. Si trop de cibles ont pour objet une même instruction (instruction ADD par exemple), les comportements provoqués dans cette instruction risquent de prendre une trop grande importance dans le calcul des métriques notamment (l'importance relative des modèles couvrant les comportements de l'instruction ADD augmenterait au détriment des autres modèles).

Deuxième ensemble : VerifyPIN

Les contextes assembleur présentés précédemment ont l'avantage de faire activer diverses structures du processeur en présentant des situations très variées. Mais il existe un risque à l'utilisation exclusive de ces contextes assembleur, celui de la non-représentativité de situations

réelles. Les modèles de faute logiciels construits par rapport aux contextes assembleur peuvent ne pas être bien adaptés aux situations rencontrées dans des applications du monde réel.

Pour pallier à ce risque, deux applications sont utilisées dans les programmes de caractérisation, en plus des contextes assembleur. Ces programmes sont utilisés ici principalement dans une optique de validation, pour vérifier que les contextes assembleurs sont suffisamment représentatifs de situations réelles. Mais ils peuvent également servir à la modélisation des fautes.

La première application est l'application VerifyPIN, déjà présentée dans la section III.4.4.1. La version 6 du programme est à nouveau utilisée, compilée avec GCC –Og et avec le PIN utilisateur fixé à 2709 et le PIN secret à 2019.

Une application réelle comme VerifyPIN n'est pas constituée d'un prologue, une cible et un épilogue comme pour les contextes assembleur ; il est donc nécessaire de définir dans quelles instructions les injections doivent avoir lieu et à quelle instruction observer les résultats de propagation. Comme l'application est relativement courte, toutes les instructions sont des cibles d'injection. Au besoin, nous distinguons les itérations d'une même instruction (une même instruction exécutée plusieurs fois dans une boucle sera considérée comme plusieurs cibles distinctes). Au total, nous obtenons 103 instructions cibles. L'observation de l'effet de propagation est quant à elle fixée à la fin du programme, quel que soit le moment d'injection.

Troisième ensemble : LittleXorKey

L'application VerifyPIN est une application très centrée sur le flot de contrôle : elle consiste principalement à faire des comparaisons, et son résultat est une valeur booléenne (authentification réussie ou non). Il est également intéressant d'utiliser une application plutôt centrée sur le flot de données. Nous proposons pour cela de développer une application simple, LittleXorKey, dont le rôle est d'appliquer l'opération XOR à deux matrices 2x2. C'est une version simplifiée d'une étape de l'algorithme de cryptographie AES. Le code de cette application est présenté en Figure 45.

```
int loopCount=0;
for(i=0 ; i < 2 ; i++){
   for(j=0 ; j < 2 ; j++){
      usr[i][j] ^= key[i][j];
      loopCount++;
   }
}
```

FIGURE 45 : CODE DE L'APPLICATION LITTLEXORKEY

Cette application est compilée avec GCC –Og. Comme pour VerifyPIN, toutes les instructions sont des cibles d'injection et l'observation de l'effet de propagation est placée à la fin du programme. Au total, nous obtenons 71 instructions cibles.

IV.6.1.2. Méthode d'injection RTL : l'ensemble M_{RTL}

L'ensemble M_{RTL} désigne les bascules du processeur qui peuvent être ciblées lors d'une injection de faute. Nous nous intéresserons ici une nouvelle fois à un processeur LowRISC déjà utilisé dans le chapitre précédent, mais plutôt que la version 0.2, nous utiliserons la version 0.4 publiée en juin 2017, qui permet d'utiliser de meilleurs scripts de simulation. La structure du pipeline est toujours la même ; ce changement de version ne rend pas obsolètes les résultats du chapitre précédent.

Les injections sont focalisées sur les bascules du pipeline. Les bascules appartenant à la mémoire du processeur ou au banc de registre ne sont pas ciblées, de même que les bascules appartenant aux étages Fetch et Decode ; pour les mêmes raisons que précédemment : nous nous intéressons avant tout au cœur du pipeline où sont réellement exécutées les instructions.

Les bascules ciblées étaient associées à un (ou plusieurs) des trois étages restants : Execute, Memory ou Write-Back ; en fonction des étages qui devaient lire l'état de ces bascules. Pour les 1116 bascules des étages EX, MEM et WB, 1308 paires bascules/étage ont été spécifiées.

La campagne d'injection RTL est constituée d'injection simples et d'injections multiples, que nous distinguerons au moment de l'analyse des résultats.

IV.6.1.3. Méthode d'injection logicielle : l'ensemble M_{SW}

Enfin, le dernier type d'entrée à spécifier pour pouvoir appliquer notre approche est l'ensemble des modèles de faute logiciels. Comme l'approche de modélisation est itérative, cet ensemble tend à évoluer : des modèles peuvent être ajoutés, modifiés ou supprimés. Le Tableau 8 liste les modèles utilisés pour ce cas d'étude. Cette liste est composée de modèles utilisés typiquement dans les analyses logicielles présentées dans le 0 et de modèles plus complexes issus de l'analyse de la microarchitecture du processeur (notamment grâce aux résultats de l'étude manuelle du 0). Les modèles non-typiques sont bâtis avant tout pour représenter les effets de fautes simple-bit dans le processeur.

La classification entre modèles typiques et non-typiques est subjective ; certains modèles considérés typiques pourraient être aussi bien considérés non-typiques, et vice versa. Le modèle **arg2_4** est considéré non-typique alors que les modèles **fwd_0** et **fwd_4** sont considérés typiques. En effet, remplacer une valeur par 0 est assez standard, mais d'autres valeurs sont plus rarement utilisées. En l'occurrence, remplacer une valeur par 4 est considéré non-typique, notamment parce que cette valeur est une des entrées d'un multiplexeur du pipeline, comme nous pouvons le voir sur la Figure 10. C'est donc l'étude de la microarchitecture qui a permis de décider que cette valeur spécifique serait utilisée. Les modèles **flipbit** (il en existe 32, de flipbit0 à flipbit31) ne sont classés dans aucune catégorie. Théoriquement, il semblerait plus pertinent de les classer dans les modèles typiques car ils représentent un effet assez générique, indépendant de l'implémentation du processeur ; mais dans la pratique, ils sont plus rarement utilisés.

TABLEAU 8 : MODELES DE FAUTE LOGICIELS DE L'ENSEMBLE M_{SW}

Nom	Description	Typique ?
Arg2_4	Remplacer le second argument de l'ALU par 4.	Non
Arg2_rs	Remplacer le deuxième argument de l'ALU par une valeur qui dépend du résultat d'un calcul précédent. Les deux bits de poids faible sont fixés à 1.	Non
Flipbit*x*	Inverser le bit *x* du résultat (il y a 32 modèles au total).	-
Fwd_0	Remplacer le premier argument de l'ALU par 0.	Oui
Fwd_1	Remplacer le premier argument de l'ALU par le résultat de la dernière instruction exécutée.	Non
Fwd_2	Remplacer le premier argument de l'ALU par le résultat de l'avant-dernière instruction exécutée.	Non
Fwd_3	Remplacer le premier argument de l'ALU par la dernière valeur lue en mémoire.	Non
Fwd_4	Remplacer le second argument de l'ALU par 0.	Oui
Fwd_5	Remplacer le second argument de l'ALU par le résultat de la dernière instruction exécutée.	Non
Fwd_6	Remplacer le second argument de l'ALU par le résultat de l'avant-dernière instruction exécutée.	Non
Fwd_7	Remplacer le second argument de l'ALU par la dernière valeur lue en mémoire.	Non
Lsb0	Remplacer le premier argument de l'ALU par une valeur qui dépend d'une instruction précédente.	Non
Res0	Le résultat de l'instruction est 0.	Oui
Skip	Sauter une instruction.	Oui
Skip_mem	Annuler le résultat de l'instruction cible après avoir exécuté l'instruction suivante.	Non
Skip_wb	Annuler le résultat de l'instruction cible après avoir exécuté les deux instructions suivantes.	Non
Test_inv	Inverser la condition d'un test.	Oui
Wrong_load	Remplacer la valeur lue en mémoire par 0.	Oui

IV.6.2. Analyses du processus d'abstraction

Une fois les campagnes d'injection réalisées, les bases de données RTL et logicielle peuvent être analysées par les différents moyens évoqués en section IV.5.2 et IV.5.3. Les analyses relatives au processus d'abstraction sont présentées dans cette section ; celles traitant de l'aspect concrétisation font l'objet de la section suivante.

IV.6.2.1. Campagne d'injections simple-bit

Commençons par analyser les résultats des injections simple-bit. Comme le nombre de bascules dans le processeur est relativement faible, les injections simple-bit ont pu être exhaustives : toutes les bascules ont été fautées, dans tous les programmes de caractérisation. Ainsi, les 1.308 paires bascule/étage ont donné lieu à 137.340 injections pour les 105 contextes assembleur, 134.724 injections pour VerifyPIN (103 instructions) et 92.868 injections pour LittleXorKey (71 instructions). A titre d'information, la campagne simple-bit exhaustive sur les contextes assembleur a duré au total 68 heures, sur un serveur de calcul (12 cœurs avec 32 Go de mémoire RAM) effectuant quatre simulations en parallèle.

Les résultats de ces campagnes sont présentées dans le Tableau 9, avec la *couverture globale GC* et la *couverture de fautes FC* des modèles de faute logiciels de M_{SW}. Pour rappel, la différence entre GC et FC est que le premier est un pourcentage relatif à la totalité des injections alors que le second est relatif au nombre de fautes analysables seulement.

TABLEAU 9 : RESULTATS DES CAMPAGNES EXHAUSTIVES D'INJECTION RTL SIMPLE-BIT

	Silencieux	Exception	Inconnu	Analysable	GC	FC
Contextes assembleur	80.6%	3.3%	0.6%	15.4%	3.7%	24.0%
VerifyPIN	65.5%	2.3%	22.4%	9.8%	2.8%	28.7%
LittleXorKey	76.6%	2.7%	2.8%	17.9%	5.2%	29.0%

Dans ce tableau, nous remarquons tout d'abord que les résultats des différents types de programmes sont assez similaires, exception faite du nombre de cas inconnus dans VerifyPIN, qui est bien plus élevé. Cela est dû à la stratégie d'observation qui n'a pu trouver l'instruction d'observation pour ces cas. Modifier la méthode d'observation ou choisir un point d'observation différent pourrait permettre de résoudre ces cas. Dans tous les cas, le taux de fautes analysables est de l'ordre de 15% (pour rappel, les fautes silencieuses ne sont pas considérées analysables car elles n'ont pas besoin d'être modélisées).

La couverture de fautes, qui est plus pertinente que la couverture globale pour comparer entre programmes, est également similaire dans les trois cas. Cela montre, en particulier, que les contextes assembleur sont suffisamment représentatifs de situations réelles. Les faibles valeurs de couverture obtenues sont discutées plus loin, en section IV.6.2.5.

Il est aussi intéressant d'étudier la couverture de divers types de modèles logiciels. Le Tableau 10 montre la couverture de modèles de faute logiciels en fonction de leur catégorie d'utilisation (typique ou non typique). Nous ne retrouvons pas les taux de couverture du Tableau 9 en additionnant simplement les valeurs du Tableau 10 car il y a des recoupements dans les modèles de faute logiciels : dans un programme de caractérisation donné, plusieurs modèles logiciels peuvent prédire les mêmes effets.

TABLEAU 10 : COUVERTURE DE FAUTES (FC) DE DIVERS TYPES DE MODELES LOGICIELS POUR LES INJECTIONS RTL SIMPLE-BIT

	Non-typique	Modèles typiques	Modèles flipbit	Typiques + flipbit
Contextes assembleur	5.2%	6.8%	15.8%	21.9%
VerifyPIN	10.1%	10.2%	19.0%	26.4%
LittleXorKey	6.6%	5.0%	23.9%	26.0%

Dans ce tableau, les modèles flipbit paraissent bien plus efficaces que les autres types de modèles. Cela est surtout dû au fait que ces modèles sont au nombre de 32 et qu'ils impactent directement le flot de données du pipeline. La couverture est à peu près également répartie entre les 32 modèles (*flipbit0* ayant la meilleure couverture). En excluant ces cas, nous remarquons que les modèles typiques et non-typiques ont une couverture similaire. Enfin, en considérant les modèles typiques et flipbit réunis, le rôle des modèles non-typiques paraît moins important, mais reste significatif. Il ne faut pas oublier, d'autre part, que les modèles typiques sont en quelque sorte « fixés », dans le sens où ce sont ceux qui sont utilisés la plupart du temps dans l'état de l'art ; alors que les modèles non-typiques résultent directement de l'analyse de la microarchitecture du processeur. Ainsi, l'importance relative des modèles non-typiques peut changer, en faveur des modèles non-typiques. Pour les contextes assembleur par exemple, les modèles de l'état de l'art représentent seulement 21,9% des comportements fautifs qu'il est possible de provoquer dans notre processeur, ce qui signifie que les autres 78,1% doivent être couverts par des modèles plus complexes.

IV.6.2.2. Campagnes d'injections multi-bit

A cause de l'explosion combinatoire induite par les injections multi-bit, des campagnes exhaustives prendraient trop de temps. Il a donc fallu se tourner vers des campagnes d'injection statistiques.

Pour ces injections, le programme de caractérisation est sélectionné aléatoirement et les bascules à fauter également, du moment qu'elles sont associées à un même étage du pipeline. Cette contrainte permet de s'assurer que les fautes sont bien injectées à un même instant de l'exécution du programme.

Plusieurs campagnes d'injection ont été menées pour chaque type de programme de caractérisation ; les fautes vont jusqu'à une multiplicité de 5. Pour chaque campagne, le nombre d'injection se situe entre 35000 et 100000. Les résultats sont présentés dans les Tableau 11, Tableau 12 et Tableau 13 avec l'intervalle de confiance de 95%, tel que défini dans [40]. Cet intervalle signifie qu'il y a 95% de chance que le taux réel soit dans l'intervalle donné dans les tableaux.

TABLEAU 11: RESULTATS DES INJECTIONS MULTI-BIT POUR LES CONTEXTES ASSEMBLEUR

	Silencieux	Exception	Inconnu	Analysable	GC	FC
1-bit	80.6%	3.3%	0.6%	15.4%	3.7%	**24.04%**
2-bit	66.2 ± 0.3%	6.4 ± 0.2%	1.2 ± 0.1%	26.1 ± 0.3%	5.9 ± 0.2%	**~ 22.51%**
3-bit	54.8 ± 0.4%	9.4 ± 0.2%	1.7 ± 0.1%	34.0 ± 0.3%	6.9 ± 0.2%	**~ 20.21%**
4-bit	46.6 ± 0.5%	12.0 ± 0.3%	2.3 ± 0.2%	39.2 ± 0.5%	7.2 ± 0.3%	**~ 18.41%**
5-bit	40.0 ± 0.5%	14.4 ± 0.4%	2.5 ± 0.2%	43.1 ± 0.5%	7.6 ± 0.3%	**~ 17.68%**

TABLEAU 12 : RESULTATS DES INJECTIONS MULTI-BIT POUR VERIFYPIN

	Silencieux	Exception	Inconnu	Analysable	GC	FC
1-bit	65.5%	2.3%	22.4%	9.8%	2.8%	**28.74%**
2-bit	54.4 ± 0.4%	4.5 ± 0.2%	24.4 ± 0.3%	16.9 ± 0.3%	4.6 ± 0.2%	**~ 27.09%**
3-bit	45.4 ± 0.4%	6.6 ± 0.2%	25.7 ± 0.3%	21.9 ± 0.3%	5.6 ± 0.2%	**~ 25.63%**
4-bit	39.3 ± 0.5%	8.4 ± 0.3%	27.1 ± 0.4%	25.2 ± 0.4%	6.1 ± 0.3%	**~ 24.18%**
5-bit	34.4 ± 0.5%	9.9 ± 0.3%	28.3 ± 0.5%	27.4 ± 0.5%	6.1 ± 0.3%	**~ 22.21%**

TABLEAU 13 : RESULTATS DES INJECTIONS MULTI-BIT POUR LITTLEXORKEY

	Silencieux	Exception	Inconnu	Analysable	GC	FC
1-bit	76.6%	2.7%	2.8%	17.9%	5.2%	**29.00%**
2-bit	59.7 ± 0.5%	5.0 ± 0.3%	5.1 ± 0.3%	30.2 ± 0.5%	7.8 ± 0.3%	**~ 25.73%**
3-bit	47.2 ± 0.6%	7.2 ± 0.3%	7.6 ± 0.3%	38.0 ± 0.5%	8.6 ± 0.3%	**~ 22.56%**
4-bit	37.4 ± 0.5%	9.8 ± 0.4%	9.4 ± 0.3%	43.5 ± 0.6%	8.9 ± 0.3%	**~ 20.34%**
5-bit	30.1 ± 0.5%	12.3 ± 0.4%	11.6 ± 0.4%	46.0 ± 0.6%	8.4 ± 0.3%	**~ 18.28%**

Ces résultats montrent que les injections multi-bit créent moins de fautes silencieuses, ce qui est assez intuitif. Plus il y a de bascules impactées dans le processeur, plus il y a de chances que cela crée des effets visibles. Une chose intéressante à remarquer est que la couverture de fautes, présentée dans la dernière colonne des tableaux, décroît plutôt linéairement avec la multiplicité des fautes. Mais les taux de couverture restent dans le même ordre que ceux obtenus dans le cas d'injections simple-bit. Ceci signifie que les modèles simple-bit restent intéressants même pour la modélisation d'injections multiples aléatoires.

Il faut cependant garder à l'esprit qu'un attaquant capable d'injecter des fautes dans des bascules bien choisies, avec des lasers par exemple, pourrait créer d'autres effets qui sortent des modèles simple-bit. Injecter des fautes multiples dans des bascules fonctionnellement dépendantes, comme évoqué dans [52], permettrait d'étudier plus précisément ces effets.

IV.6.2.3. Modèle de saut d'instruction

L'un des modèles de faute logiciels les plus utilisés est le saut d'instruction. Avec les analyses présentées, nous pouvons montrer que c'est effectivement un des modèles qui a la meilleure couverture de fautes. Cependant, comme montré dans le Tableau 8, il existe deux autres modèles de saut d'instruction, légèrement plus avancés, **skip_mem** et **skip_wb**. Ces deux modèles prennent en compte l'optimisation de forwarding. Ils se comportent de la même manière qu'un simple saut d'instruction quand il n'y a pas de dépendances entre variables, mais fonctionnent différemment dans le cas contraire. En d'autres termes, les effets de ces modèles dépendent de l'épilogue, les instructions qui suivent immédiatement l'instruction cible.

Les analyses de couverture montrent, dans le Tableau 14, que ces modèles plus avancés sont systématiquement meilleurs que le modèle typique de saut d'instruction. Notons cependant que, même s'il existe une grande redondance, chacun de ces trois modèles prédit des fautes correctes non prédites dans les autres cas.

TABLEAU 14 : COUVERTURE DE FAUTES (FC) DES MODELES SKIP, SKIP_MEM ET SKIP_WB (EN %), POUR LES CAMPAGNES D'INJECTIONS RTL SIMPLES ET MULTIPLES

Skip/skip_mem/skip_wb	1-bit	2-bit	3-bit	4-bit	5-bit
Contextes assembleur	2.8/3.3/3.3	2.9/3.6/3.5	3.1/3.6/3.5	3.0/3.5/3.4	3.2/4.2/4.0
VerifyPIN	8.0/8.6/8.4	8.2/8.7/8.5	8.1/8.8/8.5	8.5/9.1/8.8	8.3/8.8/8.5
LittleXorKey	3.2/3.5/4.2	3.0/3.4/4.5	3.0/3.3/4.1	2.9/3.2/4.0	2.7/3.0/3.9

IV.6.2.4. Couverture comportementale

Jusqu'à maintenant, la discussion a été centrée sur la couverture de fautes. Il est également intéressant de comparer cette couverture à la couverture comportementale. Le Tableau 15 expose les valeurs obtenues pour les différentes campagnes.

Ce tableau montre qu'en général, les modèles logiciels utilisés ici ont une couverture de fautes double par rapport à leur couverture comportementale. Ce résultat n'est pas surprenant car les comportements fautifs qu'un expérimentateur cherche à modéliser en priorité sont ceux qui sont observés le plus fréquemment, c'est-à-dire ceux qui ont la plus grande surface d'attaque. Ces comportements sont, en première approximation, ceux qui sont les plus faciles à provoquer pour un attaquant. Mais même si la couverture de fautes est utile, il reste important d'avoir une idée de la couverture comportementale. Dans le cas des attaques simple-bit dans VerifyPIN, si nos modèles couvrent 28,7% des fautes, cela ne représente que 14,4% des comportements fautifs possibles.

TABLEAU 15 : COMPARAISON DES COUVERTURE DE FAUTES ET COUVERTURE COMPORTEMENTALE POUR DIFFERENTES CAMPAGNES (EN %)

FC/BC	1-bit	2-bit	3-bit	4-bit	5-bit
Contextes assembleur	24.0 / 13.6	22.5 / 12.6	20.2 / 9.6	18.4 / 10.2	17.7 / 9.6
VerifyPIN	28.7 / 14.4	27.1 / 13.4	25.6 / 10.1	24.2 / 10.8	22.2 / 10.5
LittleXorKey	29.0 / 16.5	25.7 / 17.8	22.6 / 13.7	20.3 / 11.2	18.3 / 9.9

IV.6.2.5. Discussion sur les taux de couverture obtenus

Les taux de couverture obtenus peuvent sembler assez faibles malgré le nombre relativement élevé de modèles de faute logiciels utilisés. Il est cependant difficile de comparer ces taux à des travaux similaires, car les circonstances et paramètres d'injection sont souvent très différents : la stratégie d'injection et d'observation varie, tout comme le processeur, les programmes de caractérisation ou les modèles de faute logiciel et matériel.

Plusieurs raisons peuvent expliquer ces faibles taux de couverture. La première est qu'il est difficile de concevoir une stratégie d'injection et d'observation totalement satisfaisante. La complexité du

processeur ne joue pas seulement sur la difficulté à modéliser les fautes, mais également sur la difficulté à synchroniser des résultats du niveau matériel avec des résultats au niveau logiciel. Divers effets au niveau RTL peuvent ainsi parfois interférer avec la stratégie d'injection et observation : certaines instructions peuvent être rejouées, il peut y avoir des défauts de cache ou de mauvaises prédictions de branche, etc. Automatiser une stratégie d'injection et d'observation qui soit pertinente dans tous les cas est donc difficile. La stratégie utilisée ici n'étant pas parfaite, certains résultats d'injection ne sont pas corrects (dans le sens où leur comparaison avec des résultats du niveau logiciel est compromise). Cette difficulté technique ne concerne cependant qu'une faible proportion de cas et son influence sur le taux de couverture reste faible.

La deuxième difficulté à modéliser efficacement les effets des fautes au niveau logiciel tient au fait que les injections sont centrées sur le cœur du processeur (et non dans le banc de registre ou la mémoire), c'est-à-dire là où tous les calculs sont effectués. Chaque bascule peut donc avoir des effets très variés, d'autant plus que le processeur ne possède pas de contremesures matérielles.

Enfin, une dernière explication tient à la projection qui permet de passer d'un résultat d'injection à un comportement. La projection définie en section IV.5.1.2 est relativement stricte. Un résultat d'injection RTL rejoint un résultat logiciel si et seulement si l'instruction cible est la même et les effets instantanés et de propagation sont les mêmes. En considérant une projection moins stricte, de meilleures couvertures pourraient être obtenues. Nous pourrions par exemple définir la projection suivante, qui ne tient pas compte des effets instantanés :

$$\boldsymbol{\pi}'\big((c, e_1, e_2), m\big) = (c, \emptyset, e_2)$$

Avec cette projection, la couverture de faute des campagnes d'injection simple-bit passe de 28,7% à 44,2% pour VerifyPIN et de 29,0% à 49,4% pour LittleXorKey. Cette augmentation significative apparaît parce que le point d'observation est, dans ces deux cas, placé à la fin de l'application donc potentiellement bien après le point d'injection. En ignorant l'effet instantané des fautes, nous remarquons que beaucoup d'effets sont masqués avant d'arriver au point d'observation et n'ont au final pas d'impact sur l'état final du programme. La modification de la projection a cependant peu d'effet sur la couverture de fautes des contextes assembleur, qui passe de 24,0% à 24,7%. La raison est que ces programmes sont très courts (quelques instructions seulement), et que par conséquent, les deux points d'observation sont très proches. La suppression d'un de ces points n'a donc que peu d'impact sur le taux de couverture.

Le choix de la projection est une affaire de compromis. Plus il y a de points de comparaison variés (à diverses échelles de temps par rapport au point d'injection), plus les modèles logiciels devront être précis pour couvrir les fautes. Notons également que s'en tenir uniquement aux effets de propagation accroît la dépendance des résultats aux programmes de caractérisation. Plus l'observation est lointaine, moins les modèles ont besoin d'être précis et donc un taux de couverture obtenu dans un programme de caractérisation ne sera pas forcément représentatif de situations réelles. Au contraire, un modèle de faute précis à court terme a plus de chance d'avoir une couverture représentative. En considérant à la fois l'effet instantané et un effet de propagation, nous restons donc assez prudents sur l'estimation de la couverture des modèles. Pour rester cohérent avec les analyses précédentes, nous continuerons d'utiliser, dans la suite, la projection originale, **π,** qui tient compte des deux points d'observation.

Pour conclure cette section, les faibles taux de couverture obtenus sont principalement la conséquence de notre posture vis-à-vis de l'injection de fautes : notre but est d'étudier les types de

comportements possibles dans un processeur. Nous voulons donc créer un grand nombre de situations (injections uniquement dans le pipeline d'un processeur non protégé) et comprendre précisément les effets des fautes (projection stricte).

IV.6.2.6. Augmentation du taux de couverture

Après cette discussion sur les taux de couverture obtenus, la question naturelle à poser est : quelle est la difficulté à augmenter ces taux de couverture ? Est-il possible d'atteindre une couverture de 100% pour ce processeur ? Pour répondre à ces deux questions, nous considérerons uniquement les campagnes d'injection simple-bit sur les contextes assembleur (le raisonnement pour les autres campagnes étant similaire).

Avec les modèles de faute de M_{SW}, le comportement non couvert qui a la surface d'attaque la plus grande (le plus grand nombre de bascules qui, si l'une d'elle est fautée, permettent de créer ce comportement), a une surface d'attaque de 5. Si chacun des 105 contextes assembleur avait au moins un comportement avec cette même surface d'attaque, alors dans le meilleur des cas, un nouveau modèle logiciel, quel qu'il soit, ne pourrait couvrir au maximum que 525 fautes non encore couvertes. Cela signifie une augmentation de la couverture de seulement +2,5% (la couverture de faute passerait de 24,0% à 26,5%). De plus, un tel modèle serait très difficile à concevoir. Dans les modèles de M_{SW}, seuls les trois modèles de saut d'instruction couvrent plus de 400 fautes. Enfin, il y a un « rendement décroissant » dans la modélisation : plus la couverture de faute est élevée, plus il est difficile de concevoir des modèles logiciels qui couvrent beaucoup de cas non encore couverts. Cette difficulté à couvrir beaucoup de fautes avec un nombre restreint de modèles découle de la grande variété de comportements qu'il est possible de provoquer.

Ainsi, pour répondre à la première interrogation : il est possible d'augmenter la couverture de faute, mais des nouveaux modèles ne peuvent augmenter cette couverture que par petits incréments. Quant à couvrir la totalité des comportements créés au niveau RTL, la tâche n'est pas impossible mais très compliquée. Une limite basse du nombre de modèles logiciels requis peut même être extraite de la base de données RTL. Le contexte assembleur qui crée le plus de comportements analysables (ce contexte est créé à partir du prologue « fullfwd », la cible « addi » et l'épilogue « fwdboth »), génère 292 comportements analysables. Nous pouvons donc directement en conclure qu'il faudrait un minimum de 292 modèles de faute logiciels si l'on voulait couvrir parfaitement les effets des fautes pour la campagne d'injection simple-bit sur les contextes assembleur. Mais cette limite basse ne pourrait être atteinte que si chaque modèle couvrait un comportement différent et qu'en parallèle, ces mêmes modèles couvraient tous les comportements des autres contextes assembleur. Avec les modèles de M_{SW}, 37 de ces comportements sont couverts.

IV.6.2.7. Analyse de justesse des modèles logiciels

Globalement, en appliquant les modèles de M_{SW} à tous les programmes de caractérisation (contextes assembleur, VerifyPIN et LittleXorKey), nous obtenons le diagramme de justesse globale de la Figure 46. Cela correspond au final à une justesse de 76,5%, ce qui signifie que, globalement, avec les modèles de faute de M_{SW}, trois prédictions sur quatre correspondent à un comportement réel, apparu dans une simulation RTL. Dans ce diagramme, les fautes non analysables représentent une part significative des comportements créés au niveau logiciel. Dans la grande majorité des cas, cela correspond à des fautes silencieuses : le modèle n'a pas produit d'effets qui diffèrent d'une exécution non fautée du mutant.

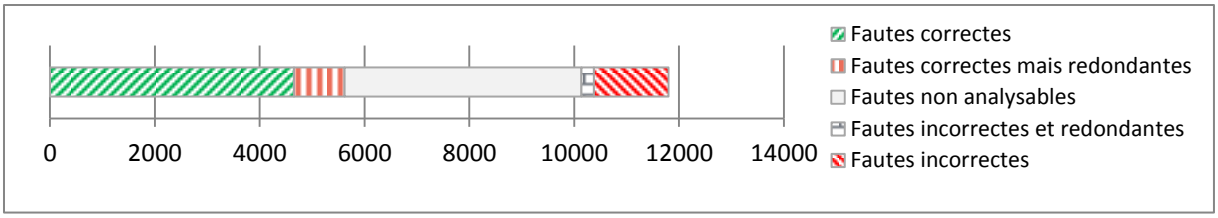

FIGURE 46 : DIAGRAMME DE JUSTESSE GLOBALE POUR LES MODELES DE FAUTE DE M_{SW}. L'AXE HORIZONTAL INDIQUE LE NOMBRE DE RESULTATS D'INJECTION LOGICIELLE.

La justesse de divers modèles de faute logiciels varie beaucoup. Les modèles **res0** et **fwd_0** ont une justesse de 100% et 96,8% respectivement, ce qui est presque parfait. Cela justifie l'emploi relativement courant de ce genre de modèle. Si dans un programme, une vulnérabilité est découverte avec l'un de ces modèles, cette vulnérabilité devrait être prise au sérieux car il est très probable qu'elle puisse être reproduite dans une injection réelle. Les modèles **fwd_1**, **skip** et **skip_mem** ont une justesse de 82,1%, 81,2% et 81,1%, respectivement, ce qui est relativement bon. Enfin, les modèles **arg2_4** et **arg2_rs** ont la plus mauvaise justesse des modèles de M_{SW}, avec des taux de 60,7% et 41,4%, respectivement. Ces modèles pourraient donc être améliorés.

IV.6.3. Analyses du processus de concrétisation

Outre les analyses d'abstraction, plusieurs types d'analyses relatives au processus de concrétisation ont aussi été définis. Leur rôle est de faire le lien des modèles de faute logiciels vers le niveau RTL. Nous allons voir que ces analyses peuvent compléter les informations données par les métriques présentées précédemment.

IV.6.3.1. Trouver où ajouter des contremesures matérielles

Pour arriver à un meilleur taux de couverture avec moins de modèles logiciels, une méthode possible est d'ajouter des contremesures matérielles. Pour décider de quelles bascules devraient être protégées en priorité, la méthode évoquée en section IV.5.3.2. est appliquée. Les résultats sont montrés en Figure 47, où nous nous limitons aux 20 bascules qui créent le plus de fautes analysables pour les injections simple-bit dans les contextes assembleur.

Sur cette figure, nous pouvons voir que, même sans considérer les modèles logiciels, aucune bascule ne se détache particulièrement du reste. L'origine des comportements est très variée. Nous remarquons également que la prise en compte des fautes couvertes par les modèles logiciels modifie l'ordre des priorités. Certaines bascules sont particulièrement bien modélisées au niveau logiciel et d'autres le sont beaucoup moins. En l'occurrence, ce sont les bascules liées à la propagation du mot d'instruction dans le pipeline qu'il semble utile de protéger en priorité.

L'étude de contremesures matérielles adaptées n'a pas été menée car en dehors du cadre de cette thèse.

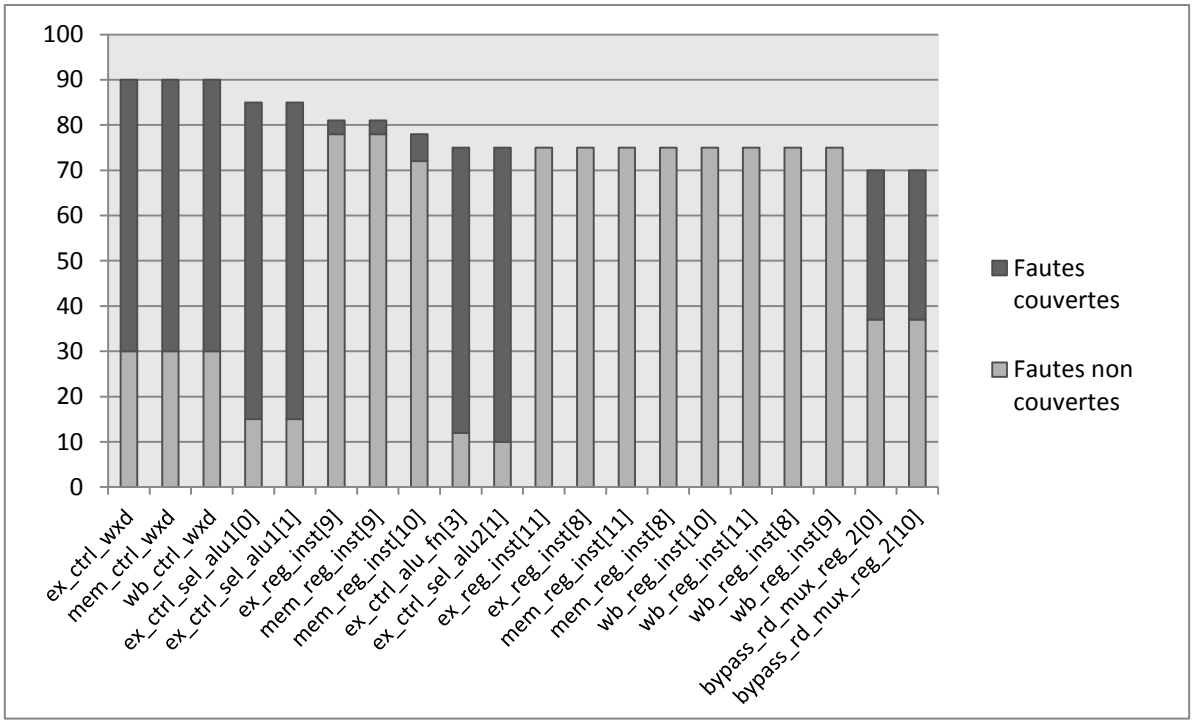

FIGURE 47: NOMBRE DE FAUTES ANALYSABLES PRODUITES EN INJECTANT DES FAUTES DANS CERTAINES BASCULES, POUR LES INJECTIONS SIMPLE-BIT DANS LES CONTEXTES ASSEMBLEUR. COMME IL Y A 105 CONTEXTES, UNE BASCULE NE PEUT CREER AU MAXIMUM QUE 105 FAUTES ANALYSABLES. LES BASCULES MONTREES ICI SONT CELLES QUI CREENT LE PLUS DE FAUTES ANALYSABLES.

IV.6.3.2. Profils des modèles

Dans la section IV.5.3.1 a été introduite la notion de *profil* de modèle pour faciliter le passage d'un modèle de faute logiciel à sa réalisation au niveau matériel. Pour rappel, un profil montre, pour un modèle de faute logiciel donné, les bascules les plus à même de recréer le même comportement lorsqu'elles sont ciblées. La Figure 48 expose les profils des modèles **arg2_4** et **skip**. Ces profils sont calculés en prenant en compte toutes les campagnes d'injections simple-bit. Les nombres montrés dans un profil ne doivent pas être comparés directement à ceux d'un autre profil car ils sont plutôt en lien avec la notion de couverture. Ce qui nous intéresse ici est la forme des profils et les différences de niveau entre les bascules d'un même profil.

La forme du profil dépend du nombre de programmes de caractérisation et de la précision du modèle en question. Un modèle très précis (qui modélise précisément un aspect du processeur) présente un pic avec une faible dispersion, alors qu'un modèle plus général a un profil plus plat car il représente un comportement plus global du processeur, non lié à une structure ou une bascule spécifique du processeur.

Le modèle **arg2_4** représente un effet très précis de la microarchitecture du processeur (modification du signal de sélection d'un multiplexeur) et son profil montre clairement la partie du processeur visée. Au contraire, le modèle **skip** est un modèle beaucoup plus générique, qui se voit aussi dans son profil : plusieurs bascules sont des candidats plausibles pour créer l'effet d'un saut d'instruction ; ce modèle n'est pas attaché à une bascule particulière. Nous pouvons tout de même voir une tendance : les bascules les plus plausibles sont des signaux de validation des étages du pipeline (« valid ») ou des signaux d'activation de l'écriture dans le banc de registre (« wxd »).

Pour les modèles de faute précis, plus le nombre de programmes de caractérisation est élevé, plus le pic doit ressortir, car ces programmes de caractérisation peuvent permettre de départager la fonction de bascules ayant un rôle similaire.

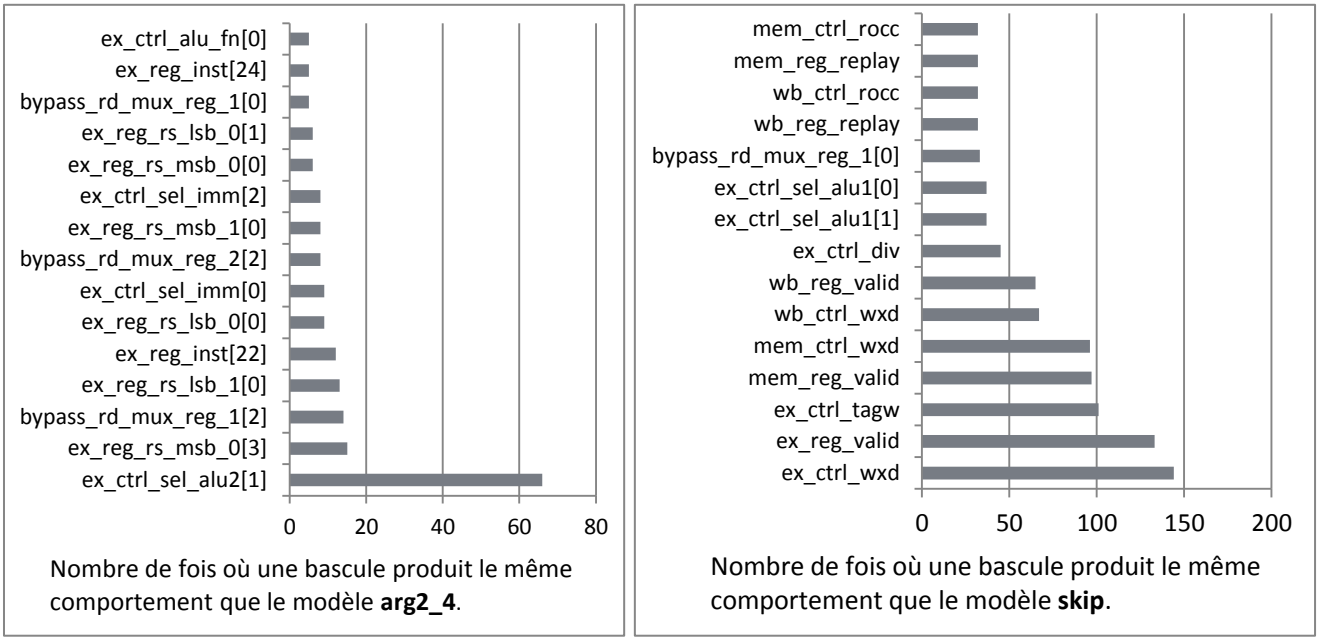

FIGURE 48 : PROFILS DES MODELES ARG2_4 ET SKIP.

IV.6.3.3. Attaques sur VerifyPIN

Comme VerifyPIN est une application réelle, qui comporte un certain nombre de contremesures, et que les injections simple-bit sont exhaustives, il est intéressant de voir si les contremesures protègent ou non contre tout type d'attaques ; et si non, si les modèles logiciels permettent de prédire ces attaques.

Dans VerifyPIN, une attaque est considérée réussie si un attaquant parvient à s'authentifier avec le mauvais code PIN et sans être détecté par une contremesure. En analysant les résultats de la campagne d'injections simple-bit, nous trouvons que parmi les 134.724 injections, 54 sont des attaques réussies. Sur ces 54 attaques, 15 sont prédites par les modèles de faute de M_{SW} (faisant écho au taux de couverture de 28,74% obtenu précédemment) ; les autres sont pour la plupart dues à des sauts très spécifiques dans l'application, qui sont difficile à modéliser dans notre outil de mutation. Certaines attaques sont prédites par plusieurs modèles.

En détail, nous trouvons que les modèles **arg2_rs** et **flipbit1** prédisent les mêmes 5 fautes, **flipbit2** prédit 4 autres fautes, **arg2_4** prédit les même 4 fautes que **flipbit2** et prédit une faute supplémentaire ; **fwd_2** prédit une autre faute, et enfin **skip** et **skip_wb** prédisent les mêmes 4 fautes.

Les modèles précédents n'ayant pas une justesse parfaite, il est utile de vérifier si les attaques prédites par ces modèles logiciels sont des attaques réelles. Par exemple, le modèle **arg2_4** a une justesse de seulement 60,7% donc les prédictions pourraient parfaitement n'être que des faux

positifs. En regardant dans son profil, dans la Figure 48, nous remarquons que la bascule la plus plausible est la bascule « ex_ctrl_sel_alu2[0] », qui en effet permet de produire une attaque.

Dans le Tableau 16 sont reportées les bascules responsables des attaques de VerifyPIN, ainsi que la place à laquelle elles apparaissent dans les profils des modèles qui prédisent ces attaques. Certaines bascules sont reportées deux fois car elles sont à l'origine d'attaques à différents moments de l'exécution de VerifyPIN. Ce tableau ne présente pas les quatre attaques prédites par les modèles de saut d'instruction, pour les raisons expliquées plus bas.

TABLEAU 16 : RANG DES BASCULES DANS LE PROFIL DES MODELES QUI PREDISENT DES ATTAQUES SUR VERIFYPIN

Attaque	arg2_4	flipbit2	arg2_rs	flipbit1	Fwd_2
ex_ctrl_sel_alu2[0]	-	-	1	12	-
ex_ctrl_sel_alu2[1] (1st)	1	-	-	-	-
ex_ctrl_sel_alu2[1] (2nd)	1	7	-	-	-
bypass_rd_mux_reg_1[1]	-	-	9	1	-
bypass_rd_mux_reg_1[2]	3	1	-	-	-
bypass_rd_mux_reg_2[1]	-	-	9	2	-
bypass_rd_mux_reg_2[2]	8	2	-	-	-
ex_reg_inst[21]	-	-	2	4	-
ex_reg_inst[22]	5	3	-	-	-
ex_reg_rs_lsb_0[1] (1st)	-	-	9	5	-
ex_reg_rs_lsb_0[1] (2nd)	-	-	-	-	4

Dans ce tableau, nous pouvons voir que pour chaque attaque réussie, la bascule responsable se trouve parmi les candidats les plus plausibles des modèles qui prédisent ces attaques, ce qui confirme l'utilité de tels profils. Une exception concerne les attaques prédites par les modèles de saut d'instruction. Les bascules responsables de ces attaques ne se retrouvent pas facilement dans leur profil. Comme évoqué précédemment, le profil de ces modèles ne fait pas apparaître clairement de structure à viser dans le processeur car ils correspondent à un effet assez global dans le processeur ; il est donc dans ce cas difficile d'incriminer une bascule particulière.

IV.7. Conclusion du chapitre

Dans ce chapitre, nous aurons mis en avant une approche permettant d'étudier les relations entre injections au niveau RTL et injections au niveau logiciel. Cette relation est bidirectionnelle avec d'une part le processus d'abstraction et d'autre part le processus de concrétisation. En réalisant des injections précises au niveau RTL et au niveau logiciel, puis en les comparant, il est possible d'obtenir des informations précieuses pour évaluer la pertinence de modèles de fautes logiciels. Ces injections sont réalisées pendant l'exécution de programmes de caractérisation qui visent à exercer diverses structures du processeur.

Pour pouvoir injecter des fautes au niveau logiciel, un outil de mutation de programme a été développé afin de représenter les comportements fautifs. Cet outil prend en entrée un fichier exécutable et un modèle de faute configurable pour générer un programme en langage C. L'un des points centraux de cet outil est d'être relativement générique, pour pouvoir accepter de nombreux modèles de faute logiciels différents. La structure du mutant généré suit les étapes matérielles d'exécution des instructions pour faciliter la tâche de modélisation.

Plusieurs méthodes d'analyse ont été développées. Pour le processus d'abstraction, des métriques de couverture et de justesse ont été proposées pour quantifier la pertinence des modèles de faute logiciels. En ce qui concerne le processus de concrétisation, le concept de profil de modèle a été évoqué, ainsi qu'une méthode pour aider à choisir quelles parties du processeur protéger. Ces méthodes peuvent se compléter pour analyser la sécurité d'un système de façon plus globale.

Ces méthodes ont ensuite été appliquées pour analyser les comportements fautifs d'un processeur. Globalement, les taux de couverture sont relativement faibles, et ceci pour plusieurs raisons. Les processeurs d'aujourd'hui sont d'une complexité très élevée, ce qui permet à une faute d'impacter son fonctionnement de plusieurs façons. Il n'est pas facile d'augmenter ce taux de couverture et une bonne couverture nécessiterait des dizaines, voire centaines de modèles de faute logiciels. La justesse des modèles testés est en revanche globalement bonne, même si certains modèles ont une justesse inférieure à 65%. Dans ces cas, leur profil peut permettre d'effectuer des simulations RTL rapides pour vérifier les résultats.

Avant de clore ce chapitre, il est également nécessaire d'évoquer quelques limitations de l'approche. Pour réaliser des campagnes d'injection qui soient reproductibles et compatibles entre niveau RTL et niveau logiciel, il a fallu faire des choix, notamment mettre de côté les effets induits par des défauts de cache ou des mauvaises spéculations. D'autre part, dans la classification des résultats d'injection, les cas inconnus n'ont pu être traités. Enfin, il est à noter que s'il est facile de refaire les analyses logicielles après avoir ajouté de nouveaux modèles, cela est moins aisé du côté matériel. Une modification dans le processeur (par exemple un ajout de contremesure) entrainerait potentiellement la nécessité de refaire les campagnes d'injection relativement longues.

Pour finir, l'approche développée ici a été appliquée pour comparer les modèles de faute logiciels à des simulations au niveau RTL. Mais l'approche est également utilisable pour d'autres méthodes d'injection, comme des méthodes d'injection réelles telles que le laser. Les modèles de faute logiciels développés dans ce cas seraient plus spécifiques au moyen d'injection utilisé, mais également plus réalistes. Les profils de modèles pourraient, plutôt que lier des bascules aux modèles logiciels, lier à ces mêmes modèles des paramètres d'injection, comme la position spatiale d'un laser ou son intensité. Le point central de l'approche développée ici réside dans la définition de méthodologies d'injections exploitables à la fois du côté matériel et du côté logiciel. Les injections et observations doivent s'effectuer aux mêmes moments.

Une fois les modèles de faute logiciels validés, ils peuvent être utilisés dans le même outil de mutation afin d'analyser la sécurité d'une application quelconque. Comme le mutant généré est écrit en langage C, ces analyses de sécurité peuvent en particulier être réalisées avec des méthodes plus avancées qu'une simple exécution. Ces types d'analyse plus avancés sont l'objet du chapitre suivant.

Chapitre V. Analyse de sécurité par analyse statique

Résumé du chapitre

Dans ce cinquième et dernier chapitre, nous explorons la possibilité d'utiliser des méthodes d'analyse statique pour analyser la sécurité de programmes modifiés grâce à notre outil de mutation. Ces méthodes doivent permettre d'évaluer plus efficacement la robustesse d'un programme par rapport à une simple exécution du mutant. Un second objectif est de s'assurer que nos mutants sont compatibles avec des outils d'analyse statique existants. Deux méthodes, basées respectivement sur l'interprétation abstraite et sur l'exécution symbolique, sont discutées, puis appliquées sur le programme VerifyPIN. Nous nous intéressons à leurs avantages et inconvénients, ainsi qu'à leur relation avec la structure du mutant.

Sommaire

V.1. Introduction du chapitre

V.1.1. Limites d'une analyse simple par exécution

Dans le chapitre précédent, nous avons développé une approche pour modéliser précisément des fautes au niveau logiciel. Les modèles obtenus doivent ensuite servir à analyser la robustesse de divers programmes exécutés par le processeur. Pour effectuer ces analyses, plusieurs méthodes peuvent être utilisées.

La première consiste à simplement compiler puis exécuter le mutant généré avec l'outil de mutation, comme nous le faisions pour les injections logicielles du chapitre précédent. Cette méthode est très simple à mettre en œuvre, mais souffre de plusieurs limitations qui, tout comme les analyses au niveau RTL, réduisent la portée des conclusions que l'on pourrait tirer. L'exécution simple du mutant ne permet en effet d'effectuer qu'une expérience à la fois (que ce soit une injection simple ou multiple) et avec des données d'entrée précises. Ainsi, si une exécution du mutant peut renseigner sur la faisabilité d'une attaque dans un cas très précis, elle ne permet pas vraiment de conclure pour d'autres instants d'injection ou d'autres données d'entrée. Il faut alors répéter les expériences pour aboutir à des conclusions plus générales sur la sécurité de l'application. Mais il est rarement possible de tester toutes les possibilités.

Les modèles développés dans le chapitre précédent posent en particulier la question de savoir si des vulnérabilités peuvent apparaître sous des conditions particulières au niveau des données d'entrée du programme. Contrairement aux modèles de faute typiques, certains de nos modèles dépendent en effet beaucoup des valeurs utilisées, comme les modèles mettant en jeu le *forwarding*.

Enfin, il est utile de noter que si l'exhaustivité de l'analyse logicielle est déjà compliquée à atteindre dans le cas d'injections simples, elle l'est encore plus lorsque l'on considère les attaques multiples. L'explosion combinatoire qui en découle diminue la représentativité des conclusions tirées sur un ensemble restreint d'expériences.

V.1.2. Utilisation de méthodes formelles

Pour parvenir à des conclusions plus robustes quant à la sécurité d'un programme, des méthodes d'analyse formelles peuvent être utilisées. Certains outils ont déjà été passés en revue dans la section II.3.2. La plupart de ces analyses formelles de sécurité ciblent un modèle de faute spécifique ; le programme à analyser est instrumenté précisément avec ce modèle. Au contraire, nous disposons d'un outil de mutation plus générique, capable de représenter des modèles de faute variés. Le but de ce chapitre est d'explorer l'utilisation de certaines méthodes formelles existantes sur les mutants générés par notre outil. Nous cherchons en particulier à nous assurer que la structure de nos mutants permet aux méthodes d'analyse de conclure dans un temps raisonnable et avec une précision raisonnable.

La diversité des modèles de faute exposés dans les chapitres précédents appelle par ailleurs à l'utilisation de différentes méthodes d'analyse. Il est envisageable que certains modèles de faute ou applications soient facilement analysables par certains outils mais pas forcément par d'autres.

Chaque méthode ou outil d'analyse possède en effet ses propres forces et faiblesses en fonction de la structure du programme et des objectifs de l'analyse. Certaines méthodes arrivent mieux à analyser le flot de contrôle du programme, d'autres sont plus efficaces dans la représentation des données, etc. Comme des modèles de faute différents modifient la structure des programmes de diverses manières, avoir à disposition plusieurs moyens d'analyse pour conclure sur la sécurité d'un programme peut être intéressant. Nous voyons ici l'intérêt d'avoir choisi une représentation en langage C pour notre mutant : ce langage étant très répandu, une multitude d'outils d'analyse peuvent être considérés. Nous nous attarderons ici sur des outils basés sur l'interprétation abstraite et sur l'exécution symbolique.

V.2. Eléments généraux sur l'analyse de sécurité au niveau logiciel

V.2.1. Propriétés de sécurité

Avant d'expliciter le fonctionnement des outils, il est utile de préciser ce que nous entendons vérifier lors des analyses. Analyser la sécurité d'un programme consiste ici à vérifier que certaines propriétés de sécurité de ce programme restent vraies même en présence d'une attaque par injection de faute. Une propriété de sécurité est une condition, un état du programme qui doit impérativement être atteint ou évité lors de l'exécution de ce programme, pour que cette exécution soit considérée sécurisée. Il peut s'agir par exemple de vérifier qu'après une boucle, l'itérateur contienne bien la valeur maximale de la boucle ; ou qu'à un point du programme, une variable contienne bien une valeur dans l'intervalle voulu. Les propriétés de sécurité peuvent aussi être plus globales, comme la vérification qu'il n'est pas possible de s'authentifier avec un code PIN incorrect.

Dans l'outil de mutation développé, ces propriétés de sécurité prennent la forme d'assertions, dont la syntaxe dépend de l'outil d'analyse utilisé, insérées à certains endroits du programme et dont la validité doit être prouvée par les outils d'analyse. Afin d'insérer automatiquement ces assertions dans le mutant, l'outil de mutation accepte une troisième entrée, visible sur la Figure 49.

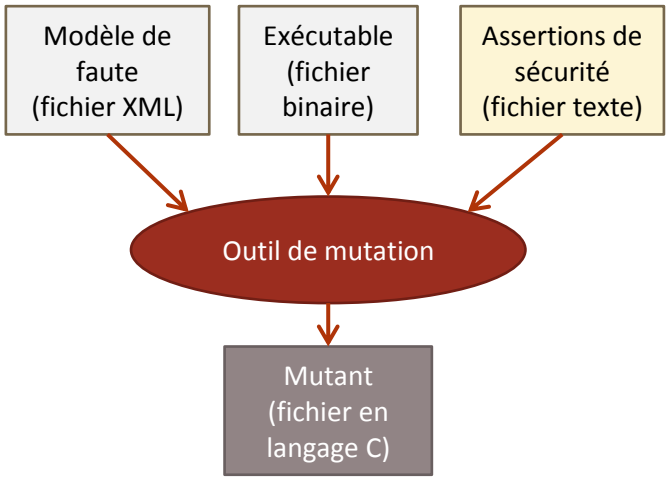

FIGURE 49 : ENTREES ET SORTIES DE L'OUTIL DE MUTATION AVEC ASSERTIONS DE SECURITE

Le fichier d'assertions est un simple document texte qui contient des lignes de code ainsi que des adresses d'instructions RISC-V où ces lignes doivent être insérées, comme le montre la Figure 50. Les lignes de code peuvent être des assertions de sécurité mais peuvent plus généralement contenir n'importe quelle série d'instructions, pourvu qu'elle soit écrite en langage C (pour être compatible avec notre mutant). Ce fichier peut donc jouer deux rôles : d'une part spécifier des propriétés à vérifier et d'autre part modifier l'exécution du programme, pour par exemple modifier les entrées du

programme sans avoir à recompiler ce programme et générer un nouveau mutant. Dans l'exemple de la Figure 50, la première ligne de code est insérée juste avant l'instruction à l'adresse 0x2ac ; elle définit la valeur d'un des digits du PIN utilisateur.

```
0x2ac :    userPin[0] = 5 ;
0x300 :    if(i != 4) attaque_reussie = 1;
```

FIGURE 50 : EXEMPLE DE FICHIER D'ASSERTIONS

Au sujet de la spécification des assertions, il est important de noter ici une petite difficulté due au fait que l'analyse est réalisée à partir du binaire du programme. Si au niveau du code source, il est facile d'ajouter une assertion pour vérifier qu'à un point donné du programme, une variable doit avoir une certaine valeur ; cela est plus compliqué au niveau du binaire où il faut à la fois trouver l'adresse de l'instruction où l'on veut placer cette assertion et trouver l'adresse mémoire ou le registre où est stockée la variable en question. Il faut donc parcourir le code désassemblé du programme pour trouver les informations nécessaires. Les outils de compilation peuvent cependant aider ; le compilateur GCC possède par exemple des options qui permettent de rapidement retrouver les informations utiles (comme le nom des fonctions, l'adresse de destination des sauts ou l'adresse où sont stockées les données en mémoire).

V.2.2. Représentation de la mémoire dans les mutants

Le choix de la structure du mutant exposée en section IV.4.1.2 (représentation des instructions RISC-V en trois phases, représentation du banc de registre sous forme de tableau, etc) joue un rôle crucial dans la capacité des outils d'analyse à offrir des conclusions pertinentes. Une mauvaise structure conduirait à des analyses trop lentes ou non conclusives. En particulier, un des points importants dont nous n'avons pas discuté est celui de la représentation de la mémoire du processeur. Plusieurs solutions peuvent être imaginées. Chacune possède ses avantages et inconvénients et certaines d'entre elles, nous le verrons, sont plus adaptées à certains types d'analyse qu'à d'autres. Aussi, plutôt que de chercher à départager les différentes méthodes, il a été décidé de laisser le choix à l'expérimentateur ; nous considérons par conséquent la représentation de la mémoire comme un paramètre du mutant. Nous exposons dans la suite trois solutions qui ont été retenues.

Dans tous les cas, les données sont découpées en octets, stockés séparément afin de coller au maximum au fonctionnement matériel. La représentation des instructions RISC-V de lecture et d'écriture a en charge de faire ce découpage/réassemblage.

V.2.2.1. Tableau long et unique

La première solution est de représenter la mémoire sous forme d'un grand tableau d'octets dont la taille est spécifiée par l'expérimentateur. Le tableau doit être assez grand pour représenter une plage d'adresses suffisante pour tout le programme à analyser. Cette méthode, illustrée en Figure 51, a pour avantages sa simplicité de mise en œuvre, ainsi que sa rapidité d'accès aux données (il y a seulement un offset entre l'adresse mémoire recherchée et la case du tableau correspondante). Mais elle présente certains inconvénients, le premier étant qu'un tableau de grande taille peut monopoliser un espace non négligeable en mémoire, ce qui peut poser des difficultés pour certains outils d'analyse et est peu intéressant quand une grande partie du tableau est inutilisée. Un autre inconvénient est qu'il est relativement peu flexible, dans la mesure où il ne permet de modéliser qu'une seule plage d'adresses ; or, un programme utilise typiquement des plages séparées pour les variables globales, la pile, etc.

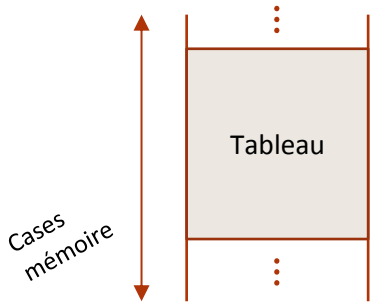

FIGURE 51 : MODELISATION DE LA MEMOIRE PAR UN TABLEAU UNIQUE

V.2.2.2. Tableaux plus restreints

La deuxième solution consiste à utiliser des tableaux de taille plus restreinte, mais en plus grand nombre. Les différents tableaux ne sont pas forcément contigus, afin d'avoir une plus grande flexibilité dans le choix des plages d'adresses modélisées. Cette méthode de modélisation est représentée en Figure 52. Elle permet d'éviter l'inconvénient de la solution précédente ; au prix d'une plus grande complexité, à deux niveaux. Tout d'abord au niveau de la spécification des plages d'adresses : cette étape est manuelle et doit faire en sorte que les plages représentées contiennent toutes les adresses utilisées par le programme. Cette contrainte est également présente dans la solution précédente, mais doit ici être gérée plus finement (plusieurs plages de taille restreinte plutôt qu'une seule plage large). Le second point à noter est que le coût à l'exécution est plus élevé car les opérations de lecture et d'écriture nécessitent une étape préliminaire de sélection du bon tableau.

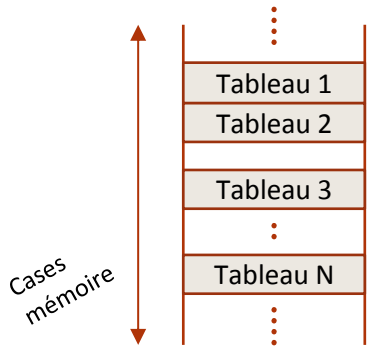

FIGURE 52 : MODELISATION DE LA MEMOIRE PAR PLUSIEURS PETITS TABLEAUX

V.2.2.3. Liste chaînée

Enfin, la dernière manière de modéliser la mémoire est de mettre en place une liste chaînée, où chaque maillon contient une adresse et un octet de donnée. Cette solution permet de s'affranchir de la difficulté à choisir les plages d'adresses à représenter : ici, n'importe quelle adresse peut être représentée dynamiquement. Du côté des inconvénients, nous retrouvons les inconvénients classiques d'une liste chaînée, notamment le temps d'exécution nécessaire à retrouver une donnée (surtout si celle-ci est située à la fin de la chaîne). Pour résumer, cette solution est simple à utiliser pour l'utilisateur (aucune configuration à effectuer), mais à un coût plus élevé pour l'analyse.

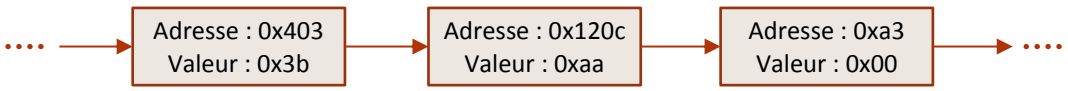

FIGURE 53 : MODELISATION DE LA MEMOIRE PAR LISTE CHAINEE

V.3. Analyse par interprétation abstraite

La première méthode d'analyse utilisée dans le cadre de cette thèse est basée sur l'interprétation abstraite. Dans cette section, nous exposons tout d'abord le principe de cette méthode d'analyse et son utilisation dans le logiciel Frama-C. Puis une analyse de VerifyPIN est effectuée. Enfin, nous concluons sur une discussion des avantages et inconvénients de la méthode.

V.3.1. Interprétation abstraite

V.3.1.1. Principe de l'interprétation abstraite

L'interprétation abstraite [17] est une méthode d'analyse statique basée sur l'abstraction de la sémantique d'une application. L'analyse est réalisée sur une abstraction des états du programme qui correspond à un sur-ensemble des états concrets de ce programme.

Ces explications étant quelque peu générales, nous pouvons nous référer à la Figure 54 pour mieux visualiser le principe. Sur cette figure, nous pouvons voir quatre courbes qui représentent les trajectoires possibles d'un programme donné. Une trajectoire est une évolution de l'état du système dans le temps. Un même programme peut avoir des trajectoires différentes en fonction par exemple des données d'entrée : avec des valeurs différentes, le programme se comporte différemment. Ces trajectoires représentent la *sémantique concrète* du programme, c'est-à-dire l'ensemble des exécutions possibles. Dans la réalité, un programme a typiquement un très grand nombre de trajectoires possibles, qui rendent l'analyse exhaustive de chaque trajectoire impraticable. C'est pourquoi, plutôt qu'analyser les trajectoires individuellement, l'interprétation abstraite propose d'abstraire la sémantique du programme pour analyser une sur-approximation de la sémantique concrète. Cette abstraction est visible sur la figure et englobe toutes les trajectoires possibles en réalité.

Sur la figure, nous représentons également trois propriétés de sécurité ; nous considérons ici que ces propriétés représentent des états que le programme doit impérativement éviter. L'analyse par interprétation abstraite consiste à chercher à savoir si la sémantique abstraite du programme recoupe ces propriétés ou non. Si la sémantique abstraite n'atteint jamais une propriété, alors nous obtenons une *preuve* que cette propriété reste vraie quelles que soient les valeurs concrètes utilisées, puisque la sémantique abstraite englobe toutes les possibilités réelles. Nous pouvons voir par exemple que sur la Figure 54, la propriété 1 est prouvée vraie.

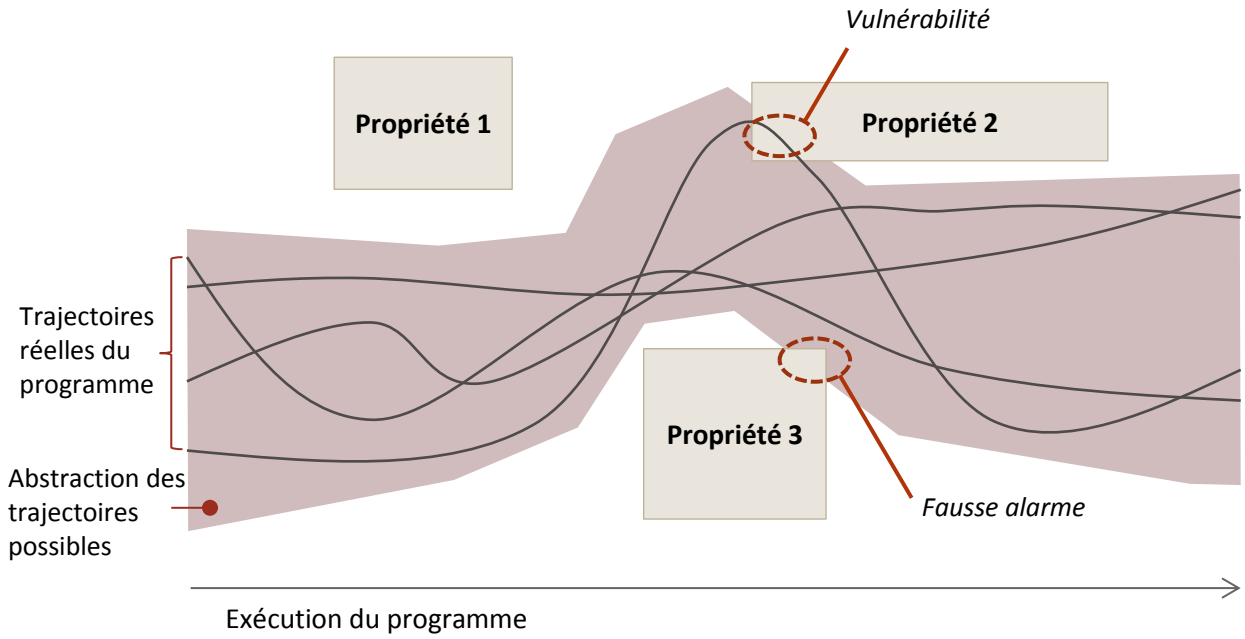

FIGURE 54 : PRINCIPE DE L'ANALYSE PAR INTERPRETATION ABSTRAITE

S'il existe un recoupement avec une propriété, en revanche, il n'est pas possible de conclure sur les exécutions réelles. Sur la figure, nous pouvons remarquer que les propriétés 2 et 3 sont toutes deux coupées par la sémantique abstraite. Cependant, s'il existe bien une vulnérabilité sur la propriété 2 (une des trajectoires réelles coupe cette propriété), il n'y en a pas pour la propriété 3, ce qui provoque une fausse alarme : l'interprétation abstraite conclut à un possible problème mais qui n'existe pas en réalité. Le calcul basé sur des sur-approximations signifie donc également qu'il n'est pas possible de générer de contre-exemples, c'est-à-dire des valeurs concrètes qui permettraient de vérifier qu'une trajectoire recoupe réellement une propriété ou non.

Ces différentes situations montrent qu'en fonction de l'abstraction choisie, les outils d'analyse peuvent soit prouver la sécurité d'un programme, soit ne pas pouvoir conclure définitivement. Dans ce second cas, un autre domaine d'abstraction (c'est-à-dire une autre représentation abstraite des états du programme) ou une autre méthode d'analyse doit être envisagée.

Il existe une infinité de domaines d'abstraction possibles. Un exemple simple de domaine est de ne tenir compte que du signe (positif, négatif ou nul) d'une variable. Les opérations du programme sont définies en accord avec l'abstraction choisie. Dans l'exemple précédent, la multiplication d'une variable positive par une variable négative donnera un résultat négatif. Par contre, l'addition d'une variable positive avec une variable négative peut donner un résultat positif, négatif ou nul ; l'analyse perd alors en précision dans ce domaine d'abstraction.

V.3.1.2. Le logiciel Frama-C

Frama-C [70] est un logiciel libre d'analyse statique pour les programmes écrits en langage C, qui permet d'utiliser de nombreuses techniques comme le calcul de préconditions, l'analyse de dépendances et bien d'autres. Une des méthodes principales est l'*analyse de valeurs*, qui est basée sur l'interprétation abstraite. Les valeurs du programme peuvent être abstraites sous forme d'intervalles, ou, si le nombre de valeurs est suffisant petit, sous forme d'une liste finie de valeurs. Ainsi, plutôt que de valider la sécurité d'une application pour des valeurs concrètes, il est possible d'utiliser tout un ensemble de valeurs en même temps. La spécification des valeurs utilisées se fait

simplement par l'utilisation d'une fonction spéciale de Frama-C, *Frama_C_interval*. Les assertions de sécurité sont quant à elles écrites dans le langage de spécification ACSL [6].

V.3.1.3. Exemple d'analyse par interprétation abstraite

Pour expliciter le fonctionnement et certaines limites de l'analyse de valeurs de Frama-C, nous proposons d'étudier le pseudo-code de la Figure 55. Dans cet exemple, plusieurs variables sont spécifiées de manière abstraite : la variable *a* peut prendre la valeur 0 ou 1, et les variables *x* et *y* peuvent prendre soit la valeur 0, soit des valeurs comprises entre 5 et 100, en fonction de la branche prise.

```
a = {0,1};

if(a) { x = {5..100}; y = 0;         }
else  { x = 0;        y =  {5..100}; }

z = x * y;
```

FIGURE 55 : PSEUDO-CODE EXEMPLE DE L'INTERPRETATION ABSTRAITE

Si l'on étudie rapidement le programme, nous remarquons que les deux branches peuvent être prises en fonction de la valeur de la variable *a*, mais dans tous les cas, une des variables *x* ou *y* a la valeur **0** ; il est alors évident que le produit de ces variables doit être nul. En analysant ce programme avec Frama-C pourtant, l'analyse déduit que le produit donne des valeurs dans l'intervalle **{0..10000}**. Cet intervalle contient bien la valeur 0, mais en contient beaucoup d'autres. Ceci est le résultat de deux types de sur-approximations.

Le premier vient de l'exécution conditionnelle : à la fin de la structure if-else, les états des deux branches sont combinés. Ainsi, la variable *x*, qui vaut **{5..100}** dans la première branche, ou **0** dans la seconde, finit, du point de vue de l'analyse, dans l'état **{0..100}**, qui englobe toutes les valeurs possibles des deux états réels possibles. Il en va de même pour la variable *y*. Non seulement les états des deux branches sont mélangés, mais ils donnent aussi naissance à des valeurs qui n'existent dans aucune des deux branches, par exemple la valeur 1. Le deuxième type de sur-approximation présent ici vient des opérations sur les intervalles. La multiplication de deux intervalles **{0..100}** donne des résultats dans **{0..10000}**. Mais dans cet intervalle résultat sont présentes des valeurs impossibles à obtenir par la multiplication des valeurs des deux intervalles originaux, comme le nombre premier 101.

Ces sur-approximations peuvent ou non interférer avec l'analyse. Si dans notre exemple le but était de prouver que le produit était forcément positif, Frama-C pourrait conclure qu'il n'y a aucun problème. A contrario, si le but avait été de prouver que le résultat était nul, Frama-C n'aurait pu conclure à cause des valeurs non nulles causées par les sur-approximations. Pour pouvoir prouver des propriétés de sécurité précises, il nous faut donc faire attention à ces approximations et choisir précautionneusement les variables à abstraire et les objectifs à atteindre, de manière à minimiser les pertes de précision qui empêchent Frama-C de conclure.

V.3.1.4. Réduction des imprécisions

Dans l'exemple précédent, il existe en réalité un moyen d'éviter la perte de précision en jouant sur un paramètre de Frama-C. Ce paramètre est le niveau de déroulement sémantique (semantic unrolling), appelé également **s-level**. Ce paramètre peut être vu comme une ressource qui peut être utilisée par Frama-C pour notamment garder des états séparés et ainsi maintenir une bonne

précision de l'analyse, au prix bien sûr du temps de calcul. Cette ressource est utilisée automatiquement à divers endroits du programme comme les structures if-else ou les boucles (dans ce second cas, le s-level permet de dérouler les itérations de la boucle afin de les analyser séparément). Une fois cette ressource épuisée, les états restants sont combinés comme dans l'exemple de la section précédente et l'analyse perd alors grandement en précision.

Comme noté dans la documentation de l'analyse de valeurs [13], il est difficile de déterminer exactement comment le **s-level** est dépensé dans l'analyse d'un programme. Cette remarque, qui tient déjà pour l'analyse de programmes « normaux » est encore plus vraie lorsque nous introduisons une faute qui impacte le fonctionnement de ce programme. L'augmentation du **s-level** permet parfois d'aboutir à des résultats plus précis, mais peut aussi augmenter inutilement le temps d'analyse. Dans tous les cas, ce paramètre ne peut conduire l'analyse à aboutir à des conclusions erronées ; il peut seulement parfois permettre de résoudre des cas où l'outil ne peut conclure.

V.3.2. Analyse de valeurs de l'application VerifyPIN

Maintenant que le fonctionnement de l'analyse de valeurs est expliqué, cette méthode peut être appliquée à un cas réel : l'application VerifyPIN (version 6, comme dans les chapitres précédents).

V.3.2.1. Préparation de l'analyse

Nous nous intéresserons ici uniquement au modèle de faute **arg2_rs**. Pour rappel, ce modèle de faute consiste à remplacer la valeur du second argument envoyé à l'ALU, par une valeur qui dépend d'une valeur précédemment utilisée, et en fixant les deux bits de poids faible à 1. Il n'est pas primordial de comprendre exactement quelle valeur est utilisée pour suivre la suite de l'exposé, mais précisons, pour le lecteur intéressé, que cette valeur est celle de la dernière donnée ayant transitée dans le registre MSB (voir Figure 12), c'est-à-dire la dernière valeur provenant du banc de registre (sans situation de forwarding).

L'application VerifyPIN, possède diverses contremesures logicielles, comme présenté dans la section III.4.4.1. L'une d'elles consiste à vérifier, après la boucle qui compare chaque digit du PIN secret avec celui entré par l'utilisateur, que l'itérateur de la boucle est bien égal à quatre (car il y a quatre digits dans les codes PIN utilisés). Cette contremesure permet à priori de contrer une attaque qui chercherait à sauter des tours de boucle afin de ne pas comparer tous les digits du code secret et ainsi faciliter une authentification.

L'analyse de valeurs de Frama-C est utilisée ici pour s'assurer de l'efficacité de cette contremesure. Pour cela, nous posons la propriété de sécurité suivante : après la vérification de la contremesure, soit la boucle a effectué quatre itérations, soit la contremesure a détecté une anomalie. Frama-C signalerait une violation de cette propriété s'il était possible d'effectuer un nombre différent de tours de boucles sans se faire détecter par la contremesure. Cette propriété est spécifiée en ajoutant du code C au mutant, via le fichier d'assertions discuté précédemment. Il faut ici insérer trois lignes de code : une première qui initialise un compteur au début du programme, une deuxième qui incrémente ce compteur à l'intérieur de la boucle ; et enfin une troisième qui vérifie la valeur de ce compteur juste à la fin de cette boucle. Il faut bien distinguer le rôle de ce nouveau compteur par rapport au compteur déjà présent dans l'application VerifyPIN. Ce dernier fait partie de l'application à analyser ; il peut donc être impacté par une faute et sa valeur ne plus représenter le nombre réel de tours de boucle. Au contraire, le compteur que nous ajoutons ne sert qu'à l'instrumentation, il n'est pas fauté (cela n'aurait pas d'intérêt) et représente le nombre réel de tours de boucle. Les

lignes de code ajoutées ici n'interfèrent donc pas avec le déroulement du programme muté ; elles sont utilisées pour donner de l'information aux outils d'analyse.

Pour se servir pleinement des capacités de l'analyse de valeurs Frama-C, nous chercherons en particulier à prouver que la propriété de sécurité est valable *quelles que soient les valeurs d'entrée du programme*. Nous spécifierons donc que chaque digit du code secret et de celui entré par l'utilisateur pourra avoir n'importe quelle valeur dans l'intervalle {0..9}. Ainsi, nous testerons cent millions de combinaisons à la fois (pour des codes de quatre digits). L'instant d'injection prend quant à lui un intervalle de valeurs suffisamment grand pour qu'en une seule analyse, les injections dans chaque instruction concernée par le modèle de faute logiciel puissent être considérées. Les conditions de l'analyse sont résumées en Figure 56 ; la Figure 57 permet quant à elle d'illustrer la spécification de certains paramètres de l'analyse : d'abord l'instant d'injection qui prend une valeur abstraite, puis l'assertion de sécurité à vérifier (il faut impérativement que le compteur inséré finisse à la valeur 4 ou que la contremesure soit déclenchée).

Propriété à vérifier : boucle de vérification exécutée quatre fois, ou contremesure déclenchée
Contraintes :

- Tous les digits du PIN utilisateur dans l'intervalle {0..9}
- Tous les digits du PIN secret dans l'intervalle {0..9}

FIGURE 56 : CONDITIONS DE L'ANALYSE DE VALEURS DE VERIFYPIN POUR VERIFIER QUE LA BOUCLE EST EXECUTEE QUATRE FOIS

```
0x2a8:  injection_time = Frama_C_interval(0,1000);
0x2f0:  //@ assert P_attack_successful: ((countermeasure!=0) || (my_counter==4));
[…]
```

FIGURE 57 : EXTRAIT DU FICHIER D'ASSERTIONS POUR L'ANALYSE DE VALEURS

V.3.2.2. Résultats de l'analyse

L'analyse de valeurs est réalisée avec la représentation mémoire à base d'un unique tableau d'une taille de 0x20000 octets. Une discussion de l'impact de la représentation mémoire est effectuée plus loin, en section V.3.3.3. L'analyse est effectuée sur une machine virtuelle avec deux cœurs à 2,6 GHz et 8 Go de mémoire RAM. L'analyse dure 20 secondes. Avec le modèle de faute utilisé, il y a 53 instants d'injection possibles ; c'est-à-dire qu'il y a 53 instants où l'on peut injecter une faute pendant l'exécution du programme. Pour 47 de ces instants d'injection, la propriété de sécurité est valide, ce qui signifie qu'il est *prouvé* que le modèle de faute utilisé n'invalide pas la propriété de sécurité à ces instants, quelle que soit la valeur des PIN.

Un des instants d'injection problématiques correspond à une attaque sur l'initialisation de l'itérateur de boucle ; dans la suite nous ignorerons cette attaque peu intéressante pour nous car facilement prédictible avec des modèles de faute logiciels classiques. Pour les cinq instants d'injection restants, Frama-C ne peut prouver la validité de la propriété ; mais cela ne signifie pas nécessairement qu'il y a

une vulnérabilité, il peut tout aussi bien s'agir de faux positifs. Une augmentation du s-level ne permet pas de mieux conclure dans ce cas.

Pour pouvoir décider si oui ou non il existe effectivement un problème de sécurité, il est alors nécessaire d'étudier ces cas manuellement. Malheureusement, pour ce faire, Frama-C aide peu puisqu'il ne permet pas de générer de contre-exemples. Sur les cent millions de combinaisons testées, lesquelles violent la propriété de sécurité (en supposant que ce ne soit pas une fausse alarme) ? Pour vérifier cela, il faut étudier précisément comment le modèle de faute impacte les instructions incriminées.

Les cinq instants d'injection pour lesquels il y a un doute correspondent en réalité à une seule et même instruction (le lien entre instant d'injection et instruction ciblée peut être fait facilement en affichant ces informations grâce au fichier d'assertions du mutant). Il y a deux raisons pour expliquer ce phénomène. D'une part, cette instruction fait partie de la boucle de vérification des digits des codes PIN ; elle est donc exécutée à plusieurs reprises (dans ce cas, ce sont les deux premières itérations qui sont incriminées). D'autre part, le fait d'analyser le programme avec tout un ensemble de données signifie que les instructions peuvent être atteintes à différents moments : des données d'entrée différentes ne suivent pas nécessairement le même chemin d'exécution.

L'instruction visée correspond à l'incrémentation de l'itérateur de boucle. Plutôt que d'incrémenter de 1, une autre valeur est utilisée, comme représenté sur la Figure 58. Nous savons d'après le modèle de faute logiciel que la valeur a ses deux bits de poids faible fixés à 1. Si les 62 bits de poids fort ont une valeur strictement supérieure à zéro, alors l'incrémentation se fera avec une valeur supérieure ou égale à 7, donc la boucle se terminera prématurément, mais l'attaque sera détectée par la contremesure. Si par contre les 62 bits de poids fort sont tous à zéro, alors l'incrémentation se fera avec la valeur 3 plutôt que la valeur 1, ce qui signifie que l'attaquant peut sauter deux tours de boucle. Comme la contremesure ne vérifie que la valeur finale de l'itérateur, il est alors possible de sauter soit les deuxième et troisième itérations, soit les troisième et quatrième itérations, sans être détecté.

FIGURE 58 : REPRESENTATION DE L'ATTAQUE DE VERIFYPIN AVEC LE MODELE ARG2_RS

Reste alors à savoir s'il est possible que les 62 bits de poids fort de la valeur utilisée par le modèle de faute soient tous à zéro. En examinant attentivement le code assembleur de VerifyPIN, nous pouvons remarquer que les 62 bits de poids fort sont ceux du digit du code secret qui a été examiné pendant la précédente itération. Ainsi, si le premier digit du code secret est 0, 1, 2 ou 3, alors les 62 bits de poids fort sont tous nuls et l'attaquant peut sauter les deuxième et troisième itérations. Si le deuxième digit du code secret est 0, 1, 2 ou 3, alors l'attaquant peut sauter les troisième et quatrième itérations. Dans ces cas, l'attaquant a alors une chance sur cent de s'authentifier plutôt qu'une chance sur dix mille.

Cette attaque est intéressante dans la mesure où elle ne se manifeste que dans des cas particuliers des données d'entrée. C'est là qu'une analyse exhaustive comme l'analyse de valeurs de Frama-C montre son intérêt ; si l'on s'était contenté d'exécuter le code dans des cas précis, il aurait été probable que cette vulnérabilité ne soit pas découverte.

Ainsi pour certaines valeurs du code secret, la contremesure n'est pas efficace. Une contremesure plus robuste ne se contenterait pas uniquement de vérifier la valeur de l'itérateur à la fin de la boucle, mais de surveiller son évolution à l'intérieur de la boucle. La version 7 de VerifyPIN implémente d'ailleurs ce genre de contremesure et l'analyse de valeurs de Frama-C permet effectivement de prouver que le code est alors sécurisé face à ce modèle de faute.

V.3.3. Discussion sur l'analyse de valeurs

La première analyse (sur la version 6 de VerifyPIN) de la section précédente a permis de montrer à la fois la compatibilité du mutant généré et l'intérêt de l'interprétation abstraite. La propriété de sécurité a pu être prouvée vraie pour 47 instants d'injection sur 53. Après inspection manuelle, il s'avère que cinq des analyses non concluantes étaient dues à une vulnérabilité qui n'apparaît qu'avec des valeurs spécifiques du code secret. Outre ces résultats intéressants, il est utile de discuter certains points en rapport avec cette méthode d'analyse.

V.3.3.1. Propriétés invariantes

Dans l'analyse présentée précédemment, il était possible de tester les cent millions de combinaisons de PIN à la fois. Cela n'est pas toujours possible car cela dépend notamment de la nature de la propriété de sécurité à vérifier. Si l'objectif de l'analyse avait été de montrer qu'il n'est pas possible de s'authentifier avec un mauvais code PIN, plutôt que de simplement s'assurer que la boucle s'exécute quatre fois, alors il n'aurait pas été possible de tester toutes ces combinaisons à la fois. Cette impossibilité vient du fait que cette nouvelle propriété n'est pas invariante mais dépend directement des données qui sont abstraites. Cette particularité est discutée dans cette section.

Dans l'analyse de valeurs de Frama-C, les valeurs des variables sont abstraites indépendamment les unes des autres. Le domaine d'abstraction est non-relationnel : il n'est pas possible de conserver les relations entre variables. Si l'on fixe le premier digit secret à 0, le digit correspondant de l'utilisateur peut être fixé à {1,2,3,4,5,6,7,8,9} ; et de même, en fixant le digit secret à 1, il est possible de fixer le digit correspondant de l'utilisateur à {0,2,3,4,5,6,7,8,9}. En revanche, si l'on voulait combiner les deux possibilités pour le digit secret, c'est-à-dire fixer sa valeur à {0,1}, alors le digit correspondant de l'utilisateur devrait être fixé à {0,1,2,3,4,5,6,7,8,9} et l'on perdrait donc la contrainte que les deux digits soient différents. Il n'est alors plus possible de distinguer une authentification avec un code PIN correct (digits égaux), d'une authentification avec un code PIN incorrect.

Ainsi, il n'est pas possible de vérifier la propriété « l'authentification échoue si les codes PIN sont différents » car celle-ci implique une relation entre variables, que l'analyse de valeurs ne peut traiter. Pour passer outre cette restriction, il est nécessaire de simplifier la propriété de sécurité en « l'authentification échoue », tout en s'assurant que les PIN sont forcément différents dans l'abstraction choisie. Nous pouvons choisir par exemple de fixer le premier digit du PIN secret à 0 et le digit correspondant du PIN utilisateur à {1,2,3,4,5,6,7,8,9}. Les autres digits des PIN, eux, n'ont pas de contrainte particulière. Les conditions de cette nouvelle analyse sont résumées en Figure 59. L'inconvénient de cette méthode est que l'on ne teste plus que neuf millions de combinaisons à la fois. Pour pouvoir prouver que la propriété est vraie pour n'importe quelles données d'entrée, il est alors nécessaire de répéter les expériences en déplaçant la contrainte sur les digits. Dans notre cas, il faudrait répéter l'expérience quarante fois (il faut contraindre la valeur d'un digit à une valeur précise ; or un digit peut prendre dix valeurs différentes, et il y a quatre digits dans un code PIN), ce qui limite l'intérêt de la méthode.

Propriété à vérifier : authentification impossible ou déclenchement de la contremesure
Contraintes :

- Trois digits du PIN utilisateur dans l'intervalle {0..9} et un digit à une valeur fixée
- Trois digits du PIN secret dans l'intervalle {0..9} et un digit dans l'intervalle {0..9} en excluant la valeur fixée dans le PIN utilisateur.

FIGURE 59 : CONDITIONS DE L'ANALYSE DE VALEURS DE VERIFYPIN POUR VERIFIER LA NON-AUTHENTIFICATION

Ce qu'il faut retenir de cette section est que la pertinence de l'analyse de valeurs de Frama-C dépend en grande partie du type de propriété à valider, du choix des variables abstraites et de leur état abstrait.

V.3.3.2. Durée des analyses de valeurs

Dans cette section, nous donnons quelques exemples de durées d'analyse obtenues sur une machine virtuelle avec deux cœurs dédiés (2,6GHz) et 8 Go de mémoire RAM. Vu le nombre élevé de modèles de faute logiciels issus de l'étude du chapitre précédent, il est important que les analyses soient conduites en un temps raisonnable ; sinon, analyser la sécurité d'un programme se révélerait trop coûteux.

La durée nécessaire aux analyses est très variable en fonction du modèle de faute utilisé. Pour la première analyse (vérification que la boucle est exécutée quatre fois), il a fallu une analyse de 20 secondes et un niveau de s-level d'environ 600 pour le modèle **arg2_rs**. Si l'on conduit la même analyse mais avec le modèle **fwd_2**, il faut alors 40 secondes pour conclure et un s-level d'environ 1400. Enfin, pour le modèle **skip_mem**, l'analyse prend 333 secondes, mais le niveau de s-level, fixé à 3000, n'est pas suffisant pour conserver une bonne précision de l'analyse ; les résultats s'en trouvent inexploitables. Pour résoudre ce problème, une façon de faire est de découper l'analyse : plutôt qu'abstraire l'instant d'injection, celui-ci doit être fixé à une valeur concrète, et les analyses doivent être répétées pour passer en revue tous les instants possibles d'injection. Finalement, en automatisant ces expériences, l'analyse globale avec le modèles **skip_mem** dure alors 158 secondes, et il n'y a plus de problème de s-level insuffisant.

Si l'on s'intéresse maintenant à la deuxième propriété de sécurité, étudiée dans la section précédente, les temps d'analyse sont assez similaires (pour une seule analyse, en fixant un des digits) : 15 secondes pour le modèle **arg2_rs** et un s-level de 500 ; et 26 secondes pour le modèle **fwd_2** et un s-level de 1000. De façon intéressante, pour cette propriété de sécurité, il suffit d'un s-level de 1700 et d'une durée de 83 secondes pour conclure avec le modèle **skip_mem** sans restreindre l'instant d'injection. La durée d'analyse dépend donc non seulement du modèle de faute, mais également de la propriété à valider. Il faut rappeler ici que pour la propriété d'authentification, les états des PIN avaient dû être restreints. Pour analyser exhaustivement toutes les combinaisons de PIN, il faut donc dans ce cas multiplier les temps d'analyse par quarante (en partant du principe que modifier la contrainte sur les valeurs des PIN ne modifie pas trop la durée d'analyse).

Pour finir, observons les durées d'analyse pour des PIN non plus de quatre digits, mais de sept digits. Nous reprenons ici la première propriété de sécurité, celle de la vérification du nombre d'itérations de la boucle. Pour le modèle **arg2_rs**, l'analyse prend 119 secondes pour un s-level d'environ 2700. Pour le modèle **fwd_2**, un s-level de 3000 ne suffit plus ; l'analyse prend 217 secondes mais les résultats sont inexploitables. En analysant individuellement les instants d'injection, l'analyse permet de conclure et il faut une durée totale de 227 secondes. Enfin, sans surprise, il faut également

découper les instants d'injection pour le modèle **skip_mem**. Il faut alors une durée totale de 379 secondes. Evidemment, l'accroissement du nombre de chemins d'exécution à considérer rend l'analyse plus longue : la complexité de l'application joue sur l'efficacité de l'analyse. Cependant, cet accroissement reste très mesuré lorsque l'on considère qu'avec des PIN de sept digits, il y a un million de fois plus de combinaisons à tester qu'avec des PIN de quatre digits.

Les durées d'analyse sont présentées ici dans le seul but de donner des ordres de grandeur. Il est difficile de généraliser ces résultats tant les analyses dépendent de facteurs différents : programme à analyser, modèle de faute, objectifs de l'analyse, abstraction choisie, etc.

V.3.3.3. Influence de la représentation de la mémoire

L'influence du s-level sur la qualité et la rapidité de l'analyse a déjà été soulignée précédemment. Mais il existe d'autres paramètres qui ont aussi un impact important. Parmi eux, un paramètre touche à la structure du mutant : la représentation de la mémoire utilisée dans le mutant.

Dans les analyses des sections précédentes, la représentation à base d'un unique tableau était utilisée. Ce tableau avait une taille de 0x20000 octets. Cette taille est choisie par l'expérimentateur, en fonction notamment des adresses nécessaires à l'exécution du programme. De plus, plus cette plage d'adresses est proche de la plage d'adresses réellement disponible dans le processeur, plus certains modèles de faute logiciels pourront donner des résultats. Un modèle de faute qui modifierait l'adresse de lecture ou d'écriture d'une donnée aurait par exemple besoin d'une plage assez large car des adresses non représentées conduisent à une erreur lors de l'exécution du mutant. Pour l'analyse de valeurs, une variation de la taille du tableau ne change pas la durée d'analyse. Les même temps sont par exemple obtenus avec un tableau de 0x50000 octets.

En revanche, en utilisant la solution basée sur de multiples tableaux, les analyses aboutissent aux mêmes résultats mais mettent plus de temps à conclure. Pour des tableaux de 0x1000 octets, la pénalité est d'environ 5% avec 5 tableaux, mais augmente par exemple à environ 50% pour 20 tableaux (c'est-à-dire le même nombre d'octets représentés que dans le tableau unique).

Enfin, la solution basée sur une liste chaînée n'est tout simplement pas utilisable avec l'analyse de valeurs. Comme évoqué dans la documentation de l'outil [13], l'allocation dynamique de mémoire est supportée sous un format limité ; les imprécisions s'accumulent rapidement, ce qui donne des résultats sous-optimaux pour les listes chaînées.

Ainsi, la façon de représenter la mémoire dans le mutant a un impact important sur la précision et la rapidité de l'analyse Frama-C. En particulier, seules deux des trois représentations de la mémoire sont utilisables avec l'analyse de valeurs.

V.3.3.4. Conclusion sur l'analyse de valeurs Frama-C

Pour conclure, l'analyse de valeurs de Frama-C peut être particulièrement intéressante dans certaines circonstances, pour prouver rapidement qu'une application est sécurisée quelles que soient les données d'entrée du programme. Cet objectif a une importance accentuée par les modèles de faute logiciels complexes utilisés dans cette thèse ; nous avons ainsi vu que certaines vulnérabilités n'apparaissaient que sous des conditions particulières.

La vitesse et la précision de Frama-C dépendent de beaucoup de paramètres différents, ce qui rend difficile de prédire la pertinence de la méthode pour un programme donné, avec un certain modèle de faute, certains objectifs de sécurité et certains états abstraits. De manière générale, les

applications avec beaucoup de boucles et de branchements sont difficiles à analyser pour Frama-C, et encore plus lorsque les données abstraites prennent des chemins différents dans le programme. Cette limitation n'est pas surprenante car la gestion des flots de contrôle complexes est un problème difficile à résoudre. Si le s-level n'est pas suffisant pour garder séparés les états de différentes branches ou les itérations d'une boucle, ces états peuvent être combinés, rendant l'analyse imprécise. Pour pouvoir conclure, l'analyse doit donc être effectuée sur des applications au flot de contrôle relativement peu complexe, ou sur des parties isolées d'applications plus complexes. L'efficacité de la méthode repose également sur la compétence de l'utilisateur à choisir des propriétés de sécurité pertinentes pour les variables abstraites.

Un des inconvénients de l'analyse de valeurs est qu'elle peut produire de fausses alarmes dues aux approximations induites par le choix des variables abstraites. Dans certains cas, il est alors difficile de distinguer une vulnérabilité réelle d'une fausse alarme. En outre, cette incertitude sur la véracité des alarmes émises s'accompagne de l'impossibilité pour Frama-C de produire des contre-exemples. Cela complique encore la tâche de l'utilisateur qui ne peut savoir pour quels états concrets une vulnérabilité pourrait apparaître.

Il ne faut cependant pas oublier que le but premier de l'analyse de valeurs est de prouver que des propriétés de sécurité tiennent pour les états abstraits spécifiés ; et pas de prouver qu'il existe des états concrets qui violent la propriété. En cela, l'analyse de valeurs de Frama-C est utile car elle permet d'éliminer les points d'injection qui ne créent définitivement pas de vulnérabilité (pour un modèle de faute donné). En somme, dans l'analyse présentée précédemment, le résultat principal n'est pas que sur les 53 points d'injection, 6 créent de potentielles vulnérabilités, mais que 47 n'en créent pas. D'autres analyses peuvent ensuite être utilisées pour conclure sur les 6 points d'injection restants.

V.4. Analyse par exécution symbolique

La seconde méthode d'analyse utilisée dans le cadre de cette thèse est l'exécution symbolique. Comme pour l'analyse de valeurs, nous exposerons tout d'abord le principe de cette méthode d'analyse et sa mise en œuvre dans le logiciel KLEE. Puis une analyse de VerifyPIN sera effectuée. Enfin, nous conclurons sur une discussion des avantages et inconvénients de cette méthode par rapport à l'analyse de valeurs.

V.4.1. Exécution symbolique

V.4.1.1. Principe de l'exécution symbolique

L'exécution symbolique consiste à analyser un programme en substituant aux données d'entrée des symboles représentant n'importe quelle valeur. Les opérations du programme contribuent à construire une formule qui représente l'état du programme en fonction des symboles. Lorsqu'une instruction de branchement dépendant d'un symbole est atteinte, l'exécution se scinde en deux et chaque condition de branchement est reflétée dans la formule associée. Lorsqu'une branche du programme se termine ou rencontre un bug, la formule associée à cette branche est résolue à l'aide d'un solveur de contraintes SMT (Satisfiability-Modulo Theory) afin d'associer à chaque symbole une valeur concrète permettant de suivre le même chemin d'exécution. L'un des intérêts de cette méthode est donc non seulement d'explorer tous les chemins d'exécution possibles d'un programme (s'il en existe un nombre fini), mais également de fournir des valeurs permettant de suivre ces chemins.

V.4.1.2. Le logiciel KLEE

KLEE [14] est un outil open-source d'analyse par exécution concolique, c'est à-dire qui mêle exécution concrète et exécution symbolique. Il est bâti sur l'infrastructure de compilation LLVM. Dans cette infrastructure, les programmes sont représentés dans un langage intermédiaire appelé LLVM-IR (LLVM Intermediate Representation). L'exécution symbolique offerte par KLEE interprète directement ce niveau de représentation et associe à chaque instruction une série de contraintes pour l'analyse. Le compilateur de l'infrastructure LLVM s'appelle Clang ; il est utilisé pour compiler les programmes (dans notre cas, les mutants) à analyser avec KLEE.

L'utilisation de cet outil est simple et peu de modifications sont à effectuer dans le programme à analyser. Il existe principalement deux fonctions pour paramétrer nos analyses. La première, *klee_make_symbolic*, permet de déclarer une variable symbolique, c'est-à-dire associer un symbole à une variable plutôt qu'une valeur concrète. La seconde, *klee_assume*, permet d'ajouter des contraintes sur certaines variables.

V.4.1.3. Exemple d'analyse par exécution symbolique

Pour illustrer le fonctionnement de KLEE, nous proposons une analyse simple sur le pseudo-code présenté en Figure 60. Dans cet exemple, nous posons la variable *a* comme étant symbolique et nous voulons vérifier si la variable *x* contient toujours une valeur paire (ou nulle) à la fin de l'exécution.

```
a est symbolique

if(a < 0)     x = 0;
else if(a%2)  x = 3 * a + 1;
else          x = a / 2;

assert(x%2 == 0);
```

FIGURE 60 : PSEUDO-CODE EXEMPLE DE L'EXECUTION SYMBOLIQUE

Pour ce programme, KLEE trouve quatre chemins d'exécution possibles, dont un viole l'assertion. L'outil procure les valeurs suivantes pour la variable *a*, pour générer ces différents chemins :

- a = -2147483648 ; ceci correspond à la condition du premier branchement : si *a* a une valeur strictement négative, *x* vaut zéro et l'assertion est valide. La valeur générée peut paraître étrange (elle dépend de détails d'implémentation de KLEE), mais elle suit le même chemin d'exécution que n'importe quelle autre valeur strictement négative.
- a = 1 ; ceci correspond à la condition du second branchement : si *a* est impair, *x* est pair et l'assertion est valide.
- a = 0 ; ceci correspond à la condition du dernier branchement sans violation de l'assertion : si *a* est pair et que sa moitié est également paire.
- a = 2 ; ceci correspond à la condition du dernier branchement avec violation de l'assertion : si *a* est pair et que sa moitié est impaire.

Ainsi, KLEE nous informe que seule la dernière branche peut poser problème, et donne une valeur provoquant le comportement problématique. Les valeurs générées par l'outil ne sont que des exemples parmi d'autres ; d'autres valeurs auraient tout aussi bien pu être générées. L'important est d'avoir une valeur pour chaque chemin d'exécution possible.

V.4.2. Exécution symbolique de l'application VerifyPIN

Dans cette section, nous nous tournons vers l'analyse symbolique de VerifyPIN. Le but premier est toujours de s'assurer que les mutants générés par notre outil sont analysables par exécution symbolique. Nous chercherons également à dégager les avantages et inconvénients de ce type d'analyse par rapport à celle basée sur l'analyse de valeurs.

V.4.2.1. Préparation de l'analyse

L'utilisation de l'exécution symbolique de KLEE est relativement similaire à l'utilisation de l'analyse de valeurs de Frama-C ; le fichier contenant les assertions de sécurité à vérifier est donc facilement transposable d'un système à l'autre.

Une des différences est que dans KLEE, les variables peuvent être simplement déclarées symboliques, là où Frama-C requiert un intervalle de valeurs. Une autre différence, plus importante, est que des contraintes relationnelles peuvent être spécifiées : il est possible ici de poser une contrainte qui dit qu'au moins un des digits du PIN utilisateur doit être différent du digit correspondant dans le PIN secret. Cet aspect de l'exécution symbolique permet de vérifier la propriété d'authentification de VerifyPIN en une seule fois, plutôt que de découper l'analyse en fonction de la valeur des PIN, comme précédemment.

Les conditions d'injection sont résumées en Figure 61 et un extrait du fichier d'assertions utilisé est présenté sur la Figure 62. Dans cette dernière figure, la première ligne rend l'instant d'injection symbolique et la deuxième ligne pose la contrainte qu'au moins un des digits du PIN utilisateur soit différent du digit correspondant du PIN secret.

Propriété à vérifier : échec de l'authentification ou déclenchement de la contremesure
Contraintes :

- PIN utilisateur symbolique
- PIN secret symbolique
- Au moins un digit différent entre les PIN utilisateur et secret

FIGURE 61 : CONDITIONS DE L'ANALYSE SYMBOLIQUE DE VERIFYPIN POUR VERIFIER LA NON-AUTHENTIFICATION

```
0x2a8: klee_make_symbolic(&injection_time, sizeof(injection_time), "injection_time");
0x2a8: klee_assume(user_0 != secret_0 || user_1 != secret_1 || user_2 != secret_2 || user_3 != secret_3);
[…]
```

FIGURE 62 : EXTRAIT DU FICHIER D'ASSERTIONS POUR L'ANALYSE SYMBOLIQUE

V.4.2.2. Résultats de l'analyse

Une fois l'analyse effectuée (toujours avec le modèle **arg2_rs**), nous retrouvons bien la vulnérabilité déjà exposée dans le cas de l'analyse de valeurs, mais cette vulnérabilité est bien plus simple à trouver, d'une part parce que tous les chemins sont analysés (alors que pour l'interprétation il avait fallu séparer différentes combinaisons de codes PIN) ; et d'autre part parce que des contre-exemples sont générés ; il n'y a donc pas besoin de chercher manuellement quelles valeurs pourraient violer la propriété de sécurité.

Il existe en réalité plusieurs vulnérabilités, mais pour celle exposée précédemment, qui vise l'incrémentation de l'itérateur de boucle, il existe quatre chemins possibles, pour lesquels les valeurs générées par KLEE sont les combinaisons de PIN suivantes (PIN utilisateur - PIN secret) :

- (0100-0000)
- (0010-0000)
- (0010-0000)
- (0001-0000)

Pour les deux premières combinaisons, l'injection de faute est effectuée dans le premier tour de boucle et permet de sauter les deuxième et troisième itérations. Pour les deux dernières, c'est le deuxième tour de boucle qui est impacté pour sauter les troisième et quatrième itérations. Les deuxième et troisième combinaisons de PIN apparaissent donc deux fois car elles correspondent à deux chemins d'exécution différents.

Parmi les valeurs générées par KLEE qui ne conduisent pas à une attaque réussie, il est intéressant de constater la présence de la combinaison (4010-4000). D'un point de vue de l'application VerifyPIN, cette combinaison devrait être équivalente à la combinaison (0010-0000) car seul le troisième digit diffère entre le PIN utilisateur et le PIN secret ; pourtant KLEE indique que ces combinaisons ne conduisent pas au même chemin d'exécution, la seconde créant une vulnérabilité mais pas la première. Ce phénomène tient au fait que l'attaque en question ne fonctionne que si le premier digit des PIN a une valeur strictement inférieure à 4. Sans connaître à l'avance cette attaque en revanche, cette différence a de quoi interpeler, car ces combinaisons devraient être fonctionnellement équivalentes. Cette différence peut dans ce cas aider l'expérimentateur à comprendre les conditions de déclenchement de l'attaque, et en l'occurrence, montrer que l'application n'est pas totalement « équilibrée » dans ses entrées ; conclusion qui avait dû être tirée par la seule analyse du code dans le cas de l'analyse Frama-C.

V.4.3. Discussion sur l'exécution symbolique

V.4.3.1. Avantages de l'exécution symbolique

Lorsque l'on compare les conditions de l'analyse de valeurs et de l'analyse symbolique, en Figure 59 et Figure 61, nous remarquons un premier avantage pour le côté symbolique. Comme l'outil peut prendre en compte les relations entre variables, toutes les valeurs des digits peuvent être déclarées symboliques, avec une contrainte pour qu'au moins un des digits soit différent ; alors que pour Frama-C, il était nécessaire de fixer au moins une valeur. Cette caractéristique permet une analyse complète de VerifyPIN en une fois. De ce point de vue, l'analyse symbolique est donc plus simple à mettre en œuvre.

Un autre défaut de l'analyse de valeurs était que ses approximations rendaient impossible la génération de contre-exemples. Au contraire, l'exécution symbolique permet tout à fait de générer des valeurs concrètes permettant de réaliser des attaques. Ces valeurs permettent de vérifier rapidement la réalité d'une vulnérabilité et facilite la compréhension de l'attaque.

V.4.3.2. Durée des analyses et influence de la représentation de la mémoire

Comme pour l'analyse de valeurs, la représentation de la mémoire utilisée a un impact important sur la durée des analyses. En l'occurrence, pour KLEE, la représentation mémoire constituée d'un seul grand tableau peut poser problème car les formules envoyées au solveur de contraintes SMT sont trop grandes ou trop complexes. L'accès à des tableaux de grande taille avec un index symbolique est

un problème récurrent dans les méthodes d'analyse symbolique [56], aussi il n'est pas étonnant que cette représentation ne soit pas satisfaisante.

Les Tableau 17, Tableau 18 et Tableau 19 présentent le temps mis par KLEE pour analyser verifyPIN avec les modèles de faute **arg2_rs**, **fwd_2** et **skip_mem**, respectivement, pour les différentes représentations mémoire et des PIN de quatre et sept digits. Les analyses durant plus d'une heure sont stoppées manuellement. Pour la solution du grand tableau unique, la taille était fixée à 0x20000 octets, et pour la solution des multiples tableaux, cette taille était réduite à 0x1000 octets par tableau, comme pour les analyses de valeur.

Toutes les analyses qui se terminent aboutissent aux mêmes vulnérabilités et nous remarquons que la solution utilisant plusieurs tableaux de taille réduite est la plus rapide. De façon intéressante, le nombre de tableaux n'influe quasiment pas sur le temps d'analyse. Les durées d'analyse augmentent assez vite avec le nombre de digits des PIN, c'est à-dire avec la complexité du programme analysé. Cette remarque n'est pas non plus étonnante car la complexité de l'application joue sur la complexité des contraintes que le solveur doit résoudre ; or, la résolution des contraintes est l'étape la plus coûteuse d'une exécution symbolique [43,56]. Tout comme l'analyse de valeurs, l'exécution symbolique est utilisable avant tout sur des programmes relativement simples, ou sur des extraits de programmes plus complexes. Nous remarquons cependant que l'analyse de valeurs a un meilleur passage à l'échelle que l'exécution symbolique ; dans ce dernier cas, les temps d'analyse pour des PIN de sept digits sont beaucoup plus conséquents que pour les PIN de quatre digits car le nombre de chemins d'exécution du programme grandit assez vite.

TABLEAU 17 : DUREE DE L'ANALYSE SYMBOLIQUE DE VERIFYPIN AVEC LE MODELE DE FAUTE ARG2_RS, POUR DIFFERENTES REPRESENTATIONS MEMOIRE ET DIFFERENTES TAILLES DE CODE PIN

	BIG_ARRAY	SMALL_ARRAYS	LINKED_LIST
PIN de 4 digits	784s	7s	83s
PIN de 7 digits	>3600s	145s	1571s

TABLEAU 18 : DUREE DE L'ANALYSE SYMBOLIQUE DE VERIFYPIN AVEC LE MODELE DE FAUTE FWD_2, POUR DIFFERENTES REPRESENTATIONS MEMOIRE ET DIFFERENTES TAILLES DE CODE PIN

	BIG_ARRAY	SMALL_ARRAYS	LINKED_LIST
PIN de 4 digits	1157s	23s	155s
PIN de 7 digits	>3600s	347s	2965s

TABLEAU 19 : DUREE DE L'ANALYSE SYMBOLIQUE DE VERIFYPIN AVEC LE MODELE DE FAUTE SKIP_MEM, POUR DIFFERENTES REPRESENTATIONS MEMOIRE ET DIFFERENTES TAILLES DE CODE PIN

	BIG_ARRAY	SMALL_ARRAYS	LINKED_LIST
PIN de 4 digits	8s	9s	155s
PIN de 7 digits	130s	154s	>3600s

V.4.3.3. Conclusion sur l'analyse symbolique KLEE

Pour conclure, l'exécution symbolique permise par KLEE apporte de nombreux avantages, notamment une facilité à paramétrer l'analyse (il suffit de déclarer certaines variables symboliques, contrairement à l'analyse de valeurs où le choix des données abstraites doit être bien pensé) et la possibilité de générer des valeurs permettant d'atteindre les états problématiques. Toutefois, les temps d'analyse augmentent rapidement avec la complexité du programme à analyser. Les représentations mémoire ont également des comportements bien différents, notamment en fonction du modèle de faute utilisé. Enfin, l'exécution symbolique peut ne pas se terminer si le nombre de chemins à analyser est trop grand, voire infini.

V.5. Conclusion du chapitre

Dans ce chapitre, deux méthodes d'analyse statique de sécurité ont été testées sur les mutants générés par notre outil. Pour l'étude de VerifyPIN, l'analyse symbolique a permis de conclure de manière plus efficace qu'avec l'analyse de valeurs et avec moins d'écueils (pas de sur-approximations à anticiper notamment). Cette facilité d'analyse tient pour l'essentiel en deux points. Le premier est que l'état initial du système est plus simple à spécifier : toutes les variables intéressantes ont pu être déclarées symboliques alors que déclarer toutes ces variables comme intervalles dans le cas de l'analyse de valeurs peut entrainer des sur-approximations trop importantes pour pouvoir conclure. Le second point est que l'exécution symbolique procure des valeurs qui permettent d'explorer facilement le programme et ses éventuelles vulnérabilités ; chose impossible en analyse de valeurs.

Pour autant, il serait injustifié de dire que l'exécution symbolique est nécessairement meilleure que l'analyse de valeurs. Ce sont deux méthodes différentes avec des buts différents. L'analyse de valeurs sert à prouver des propriétés dans un domaine abstrait qui sur-approxime les états concrets d'un programme ; l'exécution symbolique, elle, explore les chemins d'exécution possibles et génère des valeurs test correspondant à ces chemins. Cette différence d'approche joue notamment sur le passage à l'échelle, plus favorable à l'analyse de valeurs.

Nous concluons donc que la structure du mutant généré par notre outil de mutation est compatible avec divers outils d'analyse. Cela donne une nouvelle dimension de flexibilité dans l'analyse de sécurité, en plus d'avoir des modèles de faute plus variés. Comme toujours avec l'utilisation des méthodes d'analyse statique, la durée et la précision des analyses dépendent pour beaucoup du programme à analyser et d'autres paramètres, notamment dans notre cas, du modèle de faute logiciel. Différentes représentations de la mémoire ont également été proposées pour pallier à certaines difficultés d'analyse. Nous n'avons étudié dans ce chapitre que deux méthodes d'analyse, mais la flexibilité offerte par l'outil de mutation devrait être compatible avec d'autres outils. Un certain degré d'expertise dans les outils pourrait par ailleurs permettre d'obtenir des résultats plus facilement.

Pour finir, il est utile de préciser qu'au vu de la facilité de convertir le fichier contenant les assertions, le passage de l'une à l'autre méthode est aisé. Nous pouvons donc envisager d'utiliser conjointement ces deux méthodes d'analyse, et éventuellement d'autres, pour pouvoir analyser plus facilement un programme. Une piste serait d'utiliser d'abord l'analyse de valeurs pour éliminer les cas qui ne posent pas de problème de sécurité, puis d'utiliser l'exécution symbolique sur les cas incertains.

Chapitre VI. Conclusion et perspectives

VI.1. Conclusion

L'objectif de cette thèse était de contribuer à la compréhension de l'impact logiciel d'attaques par injection de fautes matérielles dans une architecture de processeur. Cette compréhension fait quelque peu défaut aux analyses de sécurité logicielles actuelles, qui se basent sur des modèles de faute logiciels pas forcément représentatifs de situations réelles et qui conduisent par conséquent à surestimer ou sous-estimer la sécurité d'un système. En particulier, la complexité et la diversité des architectures de processeurs amènent ces difficultés de modélisation.

La première étape de ce travail a été d'étudier le principe, puis l'implémentation de plusieurs structures d'une architecture de processeur. Cette étude a permis de se rendre compte du fossé qui existe entre une vision purement logicielle d'une injection de faute et sa réalité matérielle. En particulier, l'hypothèse d'atomicité des instructions est battue en brèche par des optimisations très communes, comme l'utilisation d'un pipeline, rendant la compréhension de l'impact d'une faute difficile au niveau logiciel. Des injections de fautes au niveau RTL ont ensuite été menées en simulation, afin de faire ressortir un échantillon de comportements fautifs possibles dans un processeur d'architecture RISC-V. Pour terminer, et pour montrer l'importance d'une bonne prise en compte de ces comportements dans les analyses de sécurité logicielles, ceux-ci ont été utilisés pour casser certaines contremesures logicielles typiques, ainsi qu'une application de vérification de PIN sécurisée.

Afin de consolider les résultats de cette étude préliminaire, nous avons dans un deuxième temps défini une approche plus automatisée de la modélisation des comportements fautifs. Celle-ci a un objectif double, qui est d'abstraire les comportements matériels au niveau logiciel, tout en gardant un lien avec le matériel. Ceci permet d'étudier la sécurité d'un système à un haut niveau d'abstraction, en conservant la possibilité de revenir vers un niveau plus bas pour vérifier les résultats. Cette approche est basée sur la comparaison d'injections RTL avec des injections logicielles dans des circonstances bien définies. Pour réaliser les injections logicielles suivant des modèles de faute atypiques, un outil de mutation de programme a été développé, permettant de tester de nombreux modèles de faute différents. Dans l'approche de modélisation, des métriques de couverture et de justesse ont été définies pour évaluer la pertinence des modèles de faute logiciels ; ainsi que d'autres méthodes pour aider par exemple à la modélisation des fautes ou pour choisir quelles structures protéger au niveau matériel. Toutes ces analyses ont ensuite été appliquées dans un cas d'étude pour montrer leur intérêt.

Enfin, comme la simple modélisation logicielle des fautes ne suffit pas pour analyser la sécurité d'un système, nous avons, dans le dernier chapitre, vérifié la compatibilité de notre outil de mutation avec des méthodes d'analyse logicielle, qui permettent de conclure plus efficacement qu'avec de simples exécutions fautées. Nous avons en particulier appliqué l'analyse de valeurs de Frama-C, basée sur l'interprétation abstraite, ainsi que l'exécution symbolique permise par Klee, pour l'analyse de la sécurité de l'application de vérification de PIN. Ces études ont permis de montrer que certains modèles de faute issus du chapitre précédent permettent à un attaquant de s'authentifier. Ces attaques ne sont parfois possibles que dans des circonstances bien particulières qui seraient difficiles à détecter sans des méthodes d'analyse avancées. L'efficacité des analyses dépend de nombreux paramètres, comme la structure du mutant, le type de propriété à valider, ou le modèle de faute

considéré ; ce qui justifie l'utilisation de plusieurs outils et donc la flexibilité offerte par notre outil de mutation.

Ainsi, le travail développé dans cette thèse permet une meilleure prise en compte dans les analyses de sécurité logicielles des effets de fautes injectées dans une architecture de processeur. Pour autant, cette étude possède certaines limites qu'il est utile de rappeler. La méthode d'injection présentée ici repose sur des injections et observations très précises, adaptée à une architecture de processeur ouverte. De même, la modélisation précise des effets induits par les fautes requière une bonne connaissance du fonctionnement interne du processeur. Une deuxième limite est que dans la méthode d'injection développée, certaines injections ont été laissées de côté comme celles impliquant l'exécution spéculative (nous forcions dans notre méthode une spéculation correcte), ou celles où les points d'injection n'étaient pas trouvés ; ces cas pourraient faire l'objet d'une étude spécifique. Enfin, il faut rappeler également que les modèles de faute qui impactent directement le flot de contrôle du programme peuvent être quelque peu difficiles à utiliser dans l'outil de mutation proposé.

VI.2. Perspectives

Avant de conclure ce manuscrit, plusieurs perspectives peuvent être dégagées du travail réalisé. Outre les réponses à apporter aux limitations discutées précédemment, nous proposons ici trois pistes de réflexion, faisant respectivement écho aux trois chapitres principaux de ce manuscrit.

La première piste de réflexion est l'étude d'autres types d'architectures de processeur. Le processeur étudié dans cette thèse est doté d'une architecture relativement simple, de type RISC. Mais d'autres formes de processeurs sont possibles, comme les processeurs superscalaires à exécution dans le désordre. Ces processeurs, en intégrant des optimisations toujours plus agressives, s'éloignent toujours plus de la vision logicielle de l'exécution d'un programme et doivent par conséquent donner lieu à des comportements fautifs encore plus atypiques. De plus, l'étude de différents processeurs peut permettre de dégager certaines tendances quant à l'implémentation de différentes structures : la structure de forwarding par exemple ne sera pas implémentée de la même manière dans différents processeurs.

Une deuxième perspective se situe au niveau de l'approche de modélisation développée au Chapitre IV. Les injections RTL pourraient être modifiées pour mieux rendre compte des effets d'injections réelles. Une meilleure sélection des bascules pourrait par exemple mieux représenter les possibilités offertes par les injections multiples (nous sélectionnions ici les bascules de manière aléatoire). Une autre piste serait de remplacer les injections RTL par des injections avec des moyens réels afin de modéliser directement les effets induits dans des conditions réelles. Ceci permettrait de faire le lien entre paramètres d'injection expérimentale et modèles de faute logiciels, mais la compréhension de la propagation des fautes serait aussi moins aisée.

Enfin, pour prouver la validité de différents types de propriétés de sécurité et éventuellement étudier le cas des injections multiples, d'autres outils d'analyse de programmes pourraient être utilisés et la structure du mutant généré par notre outil pourrait être adaptée en conséquence. De nouveaux paramètres de mutation pourraient voir le jour pour aider les outils d'analyse à conclure plus efficacement sur la sécurité d'un programme. Par ailleurs, une plus grande expertise dans ces méthodes pourrait permettre de mieux cerner les difficultés et les résoudre.

Publications

Les travaux présentés dans cette thèse ont fait l'objet de plusieurs publications listées ci-dessous.

Chapitre III :

- J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "On the Importance of Analysing Microarchitecture for Accurate Software Fault Models," in 2018 21st Euromicro Conference on Digital System Design (DSD), Aug. 2018, pp. 561–564, doi: 10.1109/DSD.2018.00097.

- J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures," in 2019 Design, Automation Test in Europe Conference Exhibition (DATE), Mar. 2019, pp. 252–255, doi: 10.23919/DATE.2019.8715158.

- J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor," Microprocessors and Microsystems, vol. 71, p. 102862, Nov. 2019, doi: 10.1016/j.micpro.2019.102862.

Chapitre V :

- J. Laurent, C. Deleuze, V. Beroulle, and F. Pebay-Peyroula, "Analyzing Software Security Against Complex Fault Models with Frama-C Value Analysis," in 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), Aug. 2019, pp. 33–40, doi: 10.1109/FDTC.2019.00013.

Un dernier article a également été soumis à un journal : « Bridging the Gap between RTL and Software Fault Injection » (23 pages). Celui-ci concerne le chapitre IV.

Références

[1] Seyyed Amir Asghari, Atena Abdi, Hassan Taheri, Hossein Pedram, and Saadat Pourmozaffari. 2012. SEDSR: Soft Error Detection Using Software Redundancy. *Journal of Software Engineering and Applications* 05, 09 (September 2012), 664. DOI:https://doi.org/10.4236/jsea.2012.59078

[2] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. 2011. 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*, 105–114. DOI:https://doi.org/10.1109/FDTC.2011.9

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE* 94, 2 (February 2006), 370–382. DOI:https://doi.org/10.1109/JPROC.2005.862424

[4] Alessandro Barenghi, Guido M. Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. 2010. Low voltage fault attacks to AES. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 7–12. DOI:https://doi.org/10.1109/HST.2010.5513121

[5] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. 2010. Countermeasures against fault attacks on software implemented AES. 7. DOI:https://doi.org/10.1145/1873548.1873555

[6] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filiâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language. Retrieved from https://frama-c.com/download/acsl.pdf

[7] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. 2003. A watchdog processor to detect data and control flow errors. In *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, 144–148. DOI:https://doi.org/10.1109/OLT.2003.1214381

[8] Salma Bergaoui, Pierre Vanhauwaert, and Regis Leveugle. 2010. A New Critical Variable Analysis in Processor-Based Systems. *IEEE Transactions on Nuclear Science* 57, 4 (August 2010), 1992–1999. DOI:https://doi.org/10.1109/TNS.2010.2043540

[9] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology — CRYPTO '97* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 513–525. DOI:https://doi.org/10.1007/BFb0052259

[10] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology — EUROCRYPT '97* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 37–51. DOI:https://doi.org/10.1007/3-540-69053-0_4

[11] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. 2019. Fault attack vulnerability assessment of binary code. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems* (CS2 '19), Association for Computing Machinery, Valencia, Spain, 13–18. DOI:https://doi.org/10.1145/3304080.3304083

[12] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 16–29. DOI:https://doi.org/10.1007/978-3-540-28632-5_2

[13] David Bühler, Pascal Cuoq, Boris Yakobowski, Mathieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto. Eva - The Evolved Value Analysis plug-in. Retrieved from https://frama-c.com/download/frama-c-eva-manual.pdf

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th*

USENIX conference on Operating systems design and implementation (OSDI'08), USENIX Association, San Diego, California, 209–224.

[15] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–10.

[16] Hyungmin Cho. 2018. Impact of Microarchitectural Differences of RISC-V Processor Cores on Soft Error Effects. *IEEE Access* 6, (2018), 41302–41313. DOI:https://doi.org/10.1109/ACCESS.2018.2858773

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '77), ACM, New York, NY, USA, 238–252. DOI:https://doi.org/10.1145/512950.512973

[18] P.E. Dodd and L.W. Massengill. 2003. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science* 50, 3 (June 2003), 583–602. DOI:https://doi.org/10.1109/TNS.2003.813129

[19] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops* (SPW '15), IEEE Computer Society, Washington, DC, USA, 73–87. DOI:https://doi.org/10.1109/SPW.2015.33

[20] Louis Dureuil. 2016. Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute. phdthesis. Communauté Université Grenoble Alpes. Retrieved October 16, 2017 from https://tel.archives-ouvertes.fr/tel-01403749/document

[21] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. . 3–11. DOI:https://doi.org/10.1007/978-3-319-45477-1_1

[22] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Faber Louis-Barthelemy, Marie-Lise Flottes, Gendrier Philippe, David Hely, Régis Leveugle, Paolo Maistri, Giorgio Di Natale, Athanasios Papadimitriou, and Bruno Rouzeyre. 2018. Laser fault injection at the CMOS 28 nm technology node: an analysis of the fault model. Retrieved October 18, 2018 from https://hal-emse.ccsd.cnrs.fr/emse-01856008

[23] Jean-Max Dutertre, Jacques J.A. Fournier, Amir-Pasha Mirbaha, David Naccache, Jean-Baptiste Rigaud, Bruno Robisson, and Assia Tria. 2011. Review of fault injection mechanisms and consequences on countermeasures design. In *2011 6th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 1–6. DOI:https://doi.org/10.1109/DTIS.2011.5941421

[24] Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, and Assia Triaz. 2010. Reproducible single-byte laser fault injection. In *6th Conference on Ph.D. Research in Microelectronics Electronics*, 1–4.

[25] I.D. Elliott and I.L. Sayers. 1990. Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction. *IEE Proceedings E - Computers and Digital Techniques* 137, 1 (January 1990), 88–102.

[26] Junfeng Fan, Xu Guo, Elke De Mulder, Patrick Schaumont, Bart Preneel, and Ingrid Verbauwhede. 2010. State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 76–87. DOI:https://doi.org/10.1109/HST.2010.5513110

[27] Uriel Feige. 1998. A Threshold of Ln N for Approximating Set Cover. *J. ACM* 45, 4 (July 1998), 634–652. DOI:https://doi.org/10.1145/285055.285059

[28] D Gil, J Gracia, J. C Baraza, and P. J Gil. 2003. Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault-tolerant system. *Microelectronics Journal* 34, 1 (January 2003), 41–51. DOI:https://doi.org/10.1016/S0026-2692(02)00128-3

[29] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. 2003. Soft-error detection using control flow assertions. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, 581–588. DOI:https://doi.org/10.1109/DFTVS.2003.1250158

[30] Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz, and Ronald De Keulenaer. 2016. Efficient design and evaluation of countermeasures against fault attacks using formal verification. In *LECTURE NOTES IN COMPUTER SCIENCE*, Springer International Publishing, 177–192. DOI:http://dx.doi.org/10.1007/978-3-319-31271-2_11

[31] D. H. Habing. 1965. The Use of Lasers to Simulate Radiation-Induced Transients in Semiconductor Devices and Circuits. *IEEE Transactions on Nuclear Science* 12, 5 (October 1965), 91–100. DOI:https://doi.org/10.1109/TNS.1965.4323904

[32] W. F. Heida. 2016. Towards a fault tolerant RISC-V softcore. (2016). Retrieved January 22, 2018 from http://resolver.tudelft.nl/uuid:cee5e97b-d023-4e27-8cb6-75522528e62d

[33] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. 1994. Fault injection into VHDL models: the MEFISTO tool. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, 66–75. DOI:https://doi.org/10.1109/FTCS.1994.315656

[34] Joint Interpretation Library. 2013. Application of Attack Potential to Smartcards.

[35] M. S. Kelly, K. Mayes, and J. F. Walker. 2017. Characterising a CPU fault attack model via run-time data analysis. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 79–84. DOI:https://doi.org/10.1109/HST.2017.7951802

[36] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[37] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 388–397. DOI:https://doi.org/10.1007/3-540-48405-1_25

[38] Thomas Korak, Michael Hutter, Baris Ege, and Lejla Batina. 2014. Clock Glitch Attacks in the Presence of Heating. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 104–114. DOI:https://doi.org/10.1109/FDTC.2014.20

[39] R. Leveugle. 2000. Fault injection in VHDL descriptions and emulation. In *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 414–419. DOI:https://doi.org/10.1109/DFTVS.2000.887182

[40] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *Automation Test in Europe Conference Exhibition 2009 Design*, 502–506. DOI:https://doi.org/10.1109/DATE.2009.5090716

[41] R. Leveugle and K. Hadjiat. 2002. Multi-level fault injection experiments based on VHDL descriptions: a case study. In *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, 107–111. DOI:https://doi.org/10.1109/OLT.2002.1030192

[42] A. Lindoso, L. Entrena, M. García-Valderas, and L. Parra. 2017. A Hybrid Fault-Tolerant LEON3 Soft Core Processor Implemented in Low-End SRAM FPGA. *IEEE Transactions on Nuclear Science* 64, 1 (January 2017), 374–381. DOI:https://doi.org/10.1109/TNS.2016.2636574

[43] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. 2014. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. In *Hardware and*

Software: Verification and Testing (Lecture Notes in Computer Science), Springer International Publishing, Cham, 284–299. DOI:https://doi.org/10.1007/978-3-319-13338-6_21

[44] Paolo Maistri. 2011. Countermeasures against fault attacks: The good, the bad, and the ugly. In *2011 IEEE 17th International On-Line Testing Symposium*, 134–137. DOI:https://doi.org/10.1109/IOLTS.2011.5993825

[45] Marcel Medwed and Stefan Mangard. 2011. Arithmetic logic units with high error detection rates to counteract fault attacks. In *2011 Design, Automation Test in Europe*, 1–6. DOI:https://doi.org/10.1109/DATE.2011.5763261

[46] Amir Moradi, Mohammad T. Manzuri Shalmani, and Mahmoud Salmasizadeh. 2006. A Generalized Method of Differential Fault Attack Against AES Cryptosystem. In *Cryptographic Hardware and Embedded Systems - CHES 2006* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 91–100. DOI:https://doi.org/10.1007/11894063_8

[47] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 77–88. DOI:https://doi.org/10.1109/FDTC.2013.9

[48] Nicolas Moro. 2014. Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués. phdthesis. Université Pierre et Marie Curie - Paris VI. Retrieved October 16, 2017 from https://tel.archives-ouvertes.fr/tel-01147122/document

[49] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. 2013. Formal verification of a software countermeasure against instruction skip attacks. Retrieved March 18, 2018 from https://hal-emse.ccsd.cnrs.fr/emse-00869509

[50] B. Nicolescu, Y. Savaria, and R. Velazco. 2003. SIED: software implemented error detection. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, 589–596. DOI:https://doi.org/10.1109/DFTVS.2003.1250159

[51] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability* 51, 1 (March 2002), 111–122. DOI:https://doi.org/10.1109/24.994926

[52] Athanasios Papadimitriou. 2016. RTL Modeling of Laser Attacks for Early Evaluation of Secure ICs and Countermeasure Design. PhD Thesis. Grenoble Alpes. Retrieved October 16, 2017 from http://www.theses.fr/2016GREAT041

[53] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. S. Reorda, and L. Sterpone. 2014. A New Hybrid Nonintrusive Error-Detection Technique Using Dual Control-Flow Monitoring. *IEEE Transactions on Nuclear Science* 61, 6 (December 2014), 3236–3243. DOI:https://doi.org/10.1109/TNS.2014.2361953

[54] K. Pattabiraman, N. M. Nakka, Z. T. Kalbarczyk, and R. K. Iyer. 2013. SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework. *IEEE Transactions on Computers* 62, 11 (November 2013), 2292–2307. DOI:https://doi.org/10.1109/TC.2012.219

[55] David A. Patterson and John L. Hennessy. 2017. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.

[56] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2017), Association for Computing Machinery, New York, NY, USA, 68–78. DOI:https://doi.org/10.1145/3092703.3092728

[57] M. L. Potet, L. Mounier, M. Puys, and L. Dureuil. 2014. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, 213–222. DOI:https://doi.org/10.1109/ICST.2014.34

[58] Julien Proy, Karine Heydemann, Fabien Majéric, Albert Cohen, and Alexandre Berzati. 2019. Studying EM Pulse Effects on Superscalar Microarchitectures at ISA Level. *arXiv:1903.02623 [cs]* (March 2019). Retrieved March 12, 2019 from http://arxiv.org/abs/1903.02623

[59] N. Selmane, S. Bhasin, S. Guilley, and J.-L. Danger. 2011. Security evaluation of application-specific integrated circuits and field programmable gate arrays against setup time violation attacks. *IET Information Security* 5, 4 (December 2011), 181–190. DOI:https://doi.org/10.1049/iet-ifs.2010.0238

[60] B. Selmke, J. Heyszl, and G. Sigl. 2016. Attack on a DFA Protected AES by Simultaneous Laser Fault Injections. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 36–46. DOI:https://doi.org/10.1109/FDTC.2016.16

[61] V. Sieh, O. Tschache, and F. Balbach. 1997. VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, 32–36. DOI:https://doi.org/10.1109/FTCS.1997.614074

[62] Sung-Ming Yen and M. Joye. 2000. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers* 49, 9 (September 2000), 967–970. DOI:https://doi.org/10.1109/12.869328

[63] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl. 2013. Comprehensive analysis of software countermeasures against fault attacks. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 404–409. DOI:https://doi.org/10.7873/DATE.2013.092

[64] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens. 2017. Random Additive Signature Monitoring for Control Flow Error Detection. *IEEE Transactions on Reliability* 66, 4 (December 2017), 1178–1192. DOI:https://doi.org/10.1109/TR.2017.2754548

[65] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks, 2004*, 61–70. DOI:https://doi.org/10.1109/DSN.2004.1311877

[66] Jasper G.J. van Woudenberg, Marc F. Witteman, and Federico Menarini. 2011. Practical Optical Fault Injection on Secure Microcontrollers. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 91–99. DOI:https://doi.org/10.1109/FDTC.2011.12

[67] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont. 2016. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 47–58. DOI:https://doi.org/10.1109/FDTC.2016.21

[68] Haissam Ziade, Rafic Ayoubi, and Velazco. 2004. A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology* 1, 2 (July 2004). Retrieved March 17, 2018 from https://www.researchgate.net/publication/220413910_A_Survey_on_Fault_Injection_Techniques

[69] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. 2013. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, 110–115. DOI:https://doi.org/10.1109/IOLTS.2013.6604060

[70] Frama-C. Retrieved April 10, 2019 from https://frama-c.com/index.html

Table des figures

Liste des tableaux