Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute

Louis Dureuil

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Louis Dureuil

Thèse dirigée par **Marie-Laure Potet**

préparée au sein des laboratoires **CEA-LETI et Verimag**
et de **l'École Doctorale de Mathématiques, Sciences et Technologies de l'Information, Informatique**

Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute

Thèse soutenue publiquement le **12 octobre 2016**,
devant le jury composé de :

M. Philippe Elbaz-Vincent
Professeur à l'Université Grenoble Alpes, Président
M. Sylvain Guilley
Professeur à TELECOM-ParisTech, Rapporteur
M. Jean-Louis Lanet
Professeur à l'Université de Limoges, Rapporteur
M. Guillaume Bouffard
Ingénieur-chercheur de l'Agence Nationale de la Sécurité des Systèmes d'Information, Examinateur
M. Bruno Legeard
Professeur à l'Université de Franche-Comté, Examinateur
Mme Marie-Laure Potet
Professeur à Grenoble INP, Directeur de thèse
M. Philippe de Choudens
Evaluateur CESTI du CEA-LETI, Encadrant

Remerciements

« *No man is an island entire of itself* » ... ce manuscrit n'aurait jamais vu le jour, ou aurait été très différent, sans le soutien de nombreuses personnes. En effet, une thèse est un périple de plusieurs années, et comme pour toutes les aventures, le plus important est le voyage, pas la destination. J'aimerais donc remercier tous ceux qui ont parcouru tout ou une partie de ce chemin à mes côtés, et dont l'aide m'a permis de surmonter bien des obstacles.

Merci à Marie-Laure Potet, ma directrice de thèse, pour son investissement tout au long de ces trois années, le plaisir de travailler avec elle, et son encadrement rigoureux qui m'a poussé à donner le meilleur de moi-même.

Merci à Philippe de Choudens, mon encadrant CEA, pour sa disponibilité, son encadrement, sa bonne humeur, et ses remarques toujours pertinentes sur mon travail.

Merci à messieurs Sylvain Guilley et Jean-Louis Lanet, pour avoir accepté d'être mes rapporteurs, et pour leurs précieuses remarques qui ont contribué à améliorer le présent manuscrit.

Merci à messieurs Guillaume Bouffard, Philippe Elbaz-Vincent et Bruno Legeard pour leur présence dans mon jury, et pour avoir pris le temps de se plonger dans mes travaux.

Merci à monsieur Julien Micolod, de la DGA-MI, pour son suivi du projet SERTIF et son intérêt pour mes travaux.

Merci à tous mes collègues du CESTI, pour leur professionnalisme, leur intérêt pour la sécurité, et la bonne ambiance qu'ils contribuent à créer au laboratoire. Je remercie en particulier Jessy pour son investissement dans SERTIF et dans mes travaux, Laurent et Jessy encore pour leur aide dans les expériences réalisées sur carte, Cécile pour son aide dans la formalisation de l'inférence de modèles de faute, Charles pour nos discussions techniques, Dorian et Thomas pour avoir été mes infortunés co-bureaux, et Marie-Angela et Cécile encore pour leur relecture du présent manuscrit.

Merci à tous les amis qui m'ont soutenu tout au long de la thèse : les anciens de l'Ensimag Amélie, Benoît, Camille, Clémence, Gilles et Raphaël qui ont su rester présents, mes co-thésards Eleonora, Josselin et Maxime qui ont partagé mon sort, et Pierre pour notre échange épistolaire.

Merci enfin à ma famille, mes quatre frères Julien, Joseph, Octave et Simon pour leur soutien téléphonique, mon père pour son calme rassurant, ma mère pour ses encouragements motivants, Intel le beagle pour ses longues oreilles, et Floriane pour sa patience et son affection.

Résumé

Les composants sécurisés tels que les cartes à puce sont un type de matériel particulier qui fournit des propriétés de sécurité. Pour garantir ces propriétés contre un large éventail d'attaquants puissants, des réglementations strictes ont été adoptées, telles que les Critères Communs ou la spécification EMVCo. Des laboratoires dédiés, les CESTI, ont pour tâche d'évaluer la sécurité de ces composants dans le cadre de ces réglementations. Une partie de cette évaluation consiste à déterminer la robustesse des composants sécurisés contre les attaques par injection de faute, effectuées avec un matériel spécialisé (lasers, injecteur électromagnétique), et qui résultent en une modification du comportement du programme exécuté par le composant. Dans cette thèse, on propose une approche de bout en bout pour relier les deux phases du processus d'évaluation pour l'injection de fautes : la revue de code et les tests physiques de pénétration.

La phase de revue de code peut être assistée par l'utilisation d'un outil d'analyse de code. On propose donc un état de l'art des outils existants, en commençant par ceux du domaine voisin de la tolérance aux fautes. On constate qu'il n'existe pas d'outil couvrant les besoins spécifiques des évaluateurs CESTI.

On répond alors à la problématique spécifique de déterminer les fautes réalisables sur un certain composant grâce à l'inférence de modèles de faute. Cette nouvelle méthode introduit la notion clé de modèle de faute probabiliste, où une probabilité est attribuée aux différentes fautes en fonction du paramétrage de l'équipement d'attaque. On valide la généralité de la méthode en l'appliquant à divers composants et équipements.

On décrit ensuite CELTIC, notre outil d'analyse dynamique de code binaire capable de simuler l'injection de fautes et conçu pour répondre aux problématiques complexes des évaluateurs, telles que l'ajout du support de nouveaux jeux d'instructions. CELTIC peut utiliser les modèles de faute probabilistes pour déterminer quelles fautes injecter, et calculer de nouvelles métriques facilitant le diagnostic de robustesse des évaluateurs.

Enfin, on présente les travaux effectués sur le problème des fautes multiples indépendantes, où un attaquant peut injecter plusieurs fautes durant une seule exécution du code par le composant. Ce problème implique une forte combinatoire qui complexifie le travail de simulation des outils, ainsi que le traitement des résultats. On explore donc plusieurs pistes complémentaires pour maîtriser la combinatoire et étendre les métriques de robustesse. On détaille également l'implémentation d'un compilateur à la volée pour l'injection de fautes, permettant d'améliorer drastiquement les performances de CELTIC.

On conclut par un bilan sur l'utilisation de CELTIC au CESTI et une discussion sur l'avenir de l'injection de fautes. On décrit l'apparition de techniques de plus en plus sophistiquées pour effectuer des attaques, et de contre-mesures proportionnées pour y répondre. Pour finir, on discute de la possibilité d'une application future de l'injection de fautes aux différents objets connectés.

Abstract

Secure hardware such as smartcards is a special kind of hardware that provides security properties. In order to ensure security against a broad range of powerful attackers, strict regulations have been adopted, such as Common Criteria and the EMVCo specification. Dedicated laboratories known as ITSEF evaluate the security requirements of secure hardware. Part of this evaluation assesses the robustness of secure hardware against fault injection attacks (attacks performed with specialized equipment such as lasers and electromagnetic injectors, that result in a modification of the behavior of the program). In this thesis, we propose an end-to-end approach connecting the two parts of the evaluation process against fault injection : code review, and physical penetration testing.

Code review can be aided by using code analyzers, so we provide a study of the state of the art regarding such tools, starting with those from the related field of fault tolerance. As a result, we find that no tool covers the specific needs of ITSEF evaluators.

We then tackle the specific problem of determining which faults can be implemented on a targeted secure hardware using fault model inference. This new method introduces the key notion of probabilistic fault model, where a probability is associated to the various kinds of faults as a function of the parameters of the attack equipment. Our approach is backed up by experiments on several pieces of hardware and attack equipments.

Next, we describe CELTIC, our dynamic binary code analyzer, able to simulate fault injection, and designed to meet the needs of the evaluators, such as the ability to easily add new instruction sets. CELTIC can use the probabilistic fault models to determine which faults can be injected, and to compute new metrics that assist the evaluator into rating the robustness of the secure hardware.

Lastly, we present our work on multiple fault attacks, where an attacker has the ability to inject several faults over the course of a single execution of the code on hardware. This attack implies huge combinatorics, which makes the simulation by tools and the processing of the results more complex. Thus, we explore several complementary approaches to handle this complexity and extend the metrics of robustness. We also detail the implementation of a just in time compiler for fault injection, which allows to drastically improve the performances of CELTIC.

We conclude with a report on the usage of CELTIC at the ITSEF, and a discussion on the future of fault injection. We describe the emergence of increasingly sophisticated attack equipment, and the accordingly tailored countermeasures. Our final word discusses the possibility for future applications of fault injection attacks to Internet of Things devices.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Contexte et motivations

> Je crois que deux et deux sont quatre, [...], et que quatre et quatre sont huit.
>
> *(Molière, Dom Juan)*

Résumé du chapitre

Ce chapitre définit le contexte et les motivations qui ont conduit et guidé la thèse. On commence par définir le composant sécurisé et décrire les enjeux de sécurité que la notion recouvre via l'exemple de la carte à puce, ainsi que la réponse institutionnelle à ces enjeux : le processus de certification. On présente ensuite l'attaque par perturbation, dont la conséquence est l'injection de fautes, sujet de la thèse, qu'on introduit et qu'on illustre avec l'exemple VerifyPIN. Enfin, on parcourt les objectifs poursuivis pendant la thèse pour ouvrir sur les problématiques de recherche ainsi que sur l'approche choisie pour y répondre, et les contributions qui en ont résulté.

Sommaire

1.1. Composants sécurisés

1.1.1. Définition

Un composant sécurisé est un élément matériel qui fournit des propriétés de sécurité. L'éventail de ces propriétés couvre la *confidentialité* (le bien protégé est un secret tenu hors de la connaissance des tiers), l'*intégrité* (le bien protégé ne peut pas être modifié par un tiers), l'*authentification* (processus permettant de vérifier qu'un utilisateur connaît un secret), la *non-répudiation* (un utilisateur d'un système ne peut pas nier les actions qu'il a effectuées sur le système), la *disponibilité* (le système doit fonctionner pour les utilisateurs autorisés lorsqu'ils en ont besoin) et l'*autorisation* (processus, comme par exemple le contrôle d'accès, permettant de déterminer si un utilisateur a le droit d'effectuer une requête spécifique). Ainsi, un coffre-fort peut être considéré comme un composant sécurisé analogique, car il assure l'intégrité (et dans une certaine mesure la confidentialité) des objets que son propriétaire y dépose.

Les révolutions qu'ont été l'électronique puis l'informatique ont suscité l'apparition d'un grand nombre de nouveaux usages, accompagnés d'un fort besoin en sécurité. Ainsi, l'explosion des communications instantanées et horizontales telles qu'Internet nécessite d'assurer la confidentialité et l'intégrité des messages échangés ; les badges de contrôle d'accès remplacent les clés dans de nombreuses situations ; l'informatisation des

FIGURE 1.1. – Le coffre-fort : un composant sécurisé analogique. Illustration par Tanemori (Hatena Fotolife) [CC BY 2.1 jp (http://creativecommons.org/licenses/by/2.1/jp/deed.en)], via Wikimedia Commons

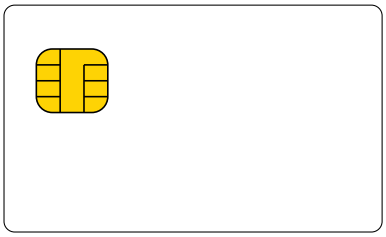


FIGURE 1.2. – Carte à puce

administrations nationales et internationales justifie les passeports biométriques ; les téléphones mobiles doivent s'authentifier auprès des réseaux des opérateurs téléphoniques pour autoriser l'utilisation du service (et identifier l'abonné rattaché à l'utilisation). La réponse à certains de ces besoins s'est matérialisée sous la forme de composants sécurisés, dont la fameuse carte à puce, qu'on présente dans la section suivante.

1.1.2. La *French touch* des composants sécurisés : la carte à puce

La carte à puce est un composant sécurisé, constitué d'un rectangle de matière plastique petit et fin (la carte) et d'un circuit intégré de silicium (la puce) qui contient un processeur et de la mémoire. La carte à puce est très facile à transporter (sa forme semble être une évolution naturelle de la carte de visite) et offre à son porteur des propriétés de sécurité même en situation de mobilité. Elle est également très peu coûteuse à produire et propose une mémoire sécurisée, c'est-à-dire une mémoire accessible seulement par la carte, et physiquement protégée contre des lectures indues de la part de l'environnement. La carte à puce a commencé comme un composant très français, porté par des chercheurs et des entreprises françaises, comme Bull, Gemplus, Axalto, Schlumberger ou Carte Bleue. Elle a ensuite été progressivement exportée à toute l'Europe, avec notamment la carte *Subscriber Identity Module* (SIM), invention allemande de 1991, puis à l'échelle mondiale.

Si la carte à puce constitue un éminent exemple de composant sécurisé, il ne faut

FIGURE 1.3. – Une télécarte. Au début des années 90, les marques utilisent les possibilités de personnalisation des cartes comme un espace publicitaire (ici, Post-it). Aujourd'hui, des sites de collectionneurs proposent l'achat et la vente de télécartes. Photo de `collection.telecarte.free.fr`.

cependant pas négliger les autres, à commencer par les divers lecteurs de cartes à puce, tels que les terminaux de paiement des commerçants, les distributeurs automatiques de billets de banques, ou les automates de vente de titres de transport. Actuellement, on trouve également les *Trusted Execution Environment* (TEE), partie sécurisée d'un processeur de téléphone portable ou de tablette, et les *Trusted Platform Module* (TPM), élément sécurisé offrant des fonctionnalités cryptographiques à une carte mère. Dans le reste de ce manuscrit, on parle souvent simplement de « carte à puce ». On se permet cet abus de langage (courant dans le domaine de recherche), et nos contributions sont valables, sauf avis expressément contraire, pour les composants sécurisés dans leur ensemble.

1.1.2.1. De la carte à mémoire à la carte à puce

L'histoire de la carte à puce commence à la fin des années 60, où le concept est ébauché entre 1968 et 1972 par plusieurs brevets sur des technologies de cartes en plastique disposant de circuits électroniques et de mémoires, déposés au Japon, en Allemagne, en Grande-Bretagne et aux États-Unis.

À ce stade, il manque encore deux éléments importants pour parler de carte à puce. En 1974, Roland Moreno dépose un brevet sur un dispositif de carte à mémoire, puis en 1977 Michel Ugon (Bull) dépose le brevet décrivant une carte disposant d'un microcontrôleur. Ces deux inventions, qui dotent la carte de sa mémoire et de ses capacités de traitement, signent la naissance de la carte à puce, la première étant produite en 1979. Dès 1983, la carte à puce connaît sa première application industrielle à l'échelle d'un pays : elle équipe les Télécartes, des cartes permettant de stocker et utiliser du crédit pour une cabine téléphonique. Le secteur bancaire français n'est pas en reste, puisque dès 1992 toutes les cartes bancaires ont leur puce. Presque 25 ans plus tard, on transporte encore plusieurs cartes à puce avec soi au quotidien. C'est le caractère pervasif de la carte à puce.

1.1.2.2. Caractère pervasif

Pervasif est un anglicisme qu'on emprunte à l'Anglais *pervasive*, lui-même dérivé du latin *per-vadere*, où la racine *vadere* signifie « aller » et le préfixe *per* signifie « à travers ». Si l'adjectif « omniprésent » constitue sans doute une traduction plus académique de *pervasive*, il n'exprime en définitive que l'idée statique d'une présence permanente en tous lieux, alors que les cartes à puce nous accompagnent au jour le jour, dans nos portefeuilles et téléphones. Pour cette raison, on a jugé que l'emprunt exprime mieux cet aspect dynamique.

Ce caractère pervasif des cartes à puce s'exprime selon au moins deux dimensions. La première est numérique. En 2014, il se serait vendu la quantité extraordinaire de 8.8 milliards de cartes à puce sur le marché [ABI15], soit un chiffre de l'ordre de la population mondiale, ce qui n'est guère étonnant quand la plupart des individus des pays développés en possèdent au moins trois (carte SIM, carte bleue, carte de santé) et doivent les changer à intervalles réguliers.

La seconde dimension pervasive s'exprime à travers l'immense variété d'applications de la carte à puce. En France, la plus évidente semble être la carte bancaire, mais en observant la situation internationale, l'exemple est moins parlant. Ainsi, les autres pays d'Europe sont moins équipés en terminaux de paiement, avec pour conséquence que les commerçants acceptent plus difficilement les cartes de débit. Au Royaume-Uni, une politique de terminal de paiement obligatoire a permis de faire baisser la fraude de 36% [Lin14], mais n'a été conduite qu'en 2006. Aux États-Unis enfin, on utilise encore le plus souvent la bande magnétique de la carte bancaire (éventuellement accompagnée d'une signature) [Lin14]. En effet, une très grande majorité d'Américains utilise essentiellement des cartes de crédit [1], où un établissement tiers de crédit sert alors d'intermédiaire entre le commerçant et le compte bancaire du particulier, ce dernier remboursant son crédit à intervalles réguliers, et laissant ainsi un délai pour pallier une fraude éventuelle.

La carte SIM, autre application particulièrement pervasive, est un succès mondial indiscutable, avec une estimation de 6,3 milliards de connexions SIM actives à la fin 2013 [GSM14]. On peut penser également aux applications dans la santé (carte Vitale) et aux passeports et autres papiers d'identité « biométriques », issus d'efforts au niveau national ou européen.

Ce caractère pervasif a cependant un revers : une faille de sécurité sur une carte répandue pourrait conduire à un fraude de grande ampleur. Il en résulte un très fort besoin en sécurité.

1.1.2.3. Besoin en sécurité

En plus du caractère pervasif des cartes à puce discuté en section précédente, le besoin en sécurité des cartes à puce se trouve renforcé par plusieurs facteurs. Ainsi, une carte à puce ne peut pas être mise à jour aisément après sa commercialisation. Si cette possibilité est techniquement réalisable (patchs), elle est difficile à mettre en œuvre sur le terrain dans le contexte des cartes à puce : comment le public pourrait avoir confiance dans une carte qu'il doit mettre à jour tous les mois ?

1. à ne pas confondre avec nos cartes de débit qui débitent directement notre compte

Par ailleurs, la carte à puce souffre de contraintes liées à sa nature de système embarqué. Sa taille et son coût réduits limitent fortement les dimensions de sa mémoire et sa puissance de calcul, ce qui exclut les protections les plus gourmandes.

Enfin, l'environnement de la carte à puce est hostile. Elle peut être attaquée physiquement, soit après un vol, soit directement par le porteur de la carte. En effet, une carte à puce contient typiquement des informations auxquelles même le porteur n'a pas accès (des clés de chiffrement par exemple) et qu'il peut avoir intérêt à forcer. On pense par exemple à la télévision à péage, indissociable de la longue histoire des décodeurs pirates. Un utilisateur peut aussi avoir intérêt à pirater sa carte vitale pour s'octroyer des remboursements à 100%, ou à faire des copies de son passeport biométrique pour les vendre.

Ce besoin renforcé en sécurité a suscité des réponses proportionnées de la part des institutions, au niveau national et international. C'est le processus de certification d'un composant sécurisé, abordé dans la section 1.1.3.

1.1.2.4. Le futur des composants sécurisés ?

On ne peut conclure ce tour d'horizon de la carte à puce sans évoquer son avenir.

La table ronde « *20 years of CARDIS, 40 years of smart cards: Where do we go from there?* » à la conférence CARDIS 2014 a permis à des experts du domaine de discuter leur idée du futur de la carte à puce, 40 ans après le brevet de Roland Moreno. Si les ventes de cartes se portent indubitablement bien, l'environnement de la carte à puce a énormément changé, et l'industrie cherche activement un successeur à la carte à puce. Cette tendance peut s'expliquer par plusieurs facteurs :

Le premier est un facteur d'intégration. La carte à puce représente aujourd'hui une industrie à part, avec ses propres règles et savoir-faire matériel et logiciel, à l'heure où les géants de la Silicon Valley ont enfin industrialisé et maîtrisé leurs processus de création logicielle. Quand Apple présente des technologies telles que l'*Apple SIM* (un module logiciel qui vise à remplacer la carte SIM par une implémentation purement logicielle) ou l'*Apple Pay* (un moyen de paiement sans contact depuis un *iphone*), l'objectif est de pouvoir utiliser l'éventail des compétences acquises dans le domaine de la conception matérielle et surtout logicielle traditionnelles. Dans une industrie en très rude compétition, éliminer le composant physique permet également de réduire un peu plus les coûts de fabrication. Enfin, cette disparition est un élément supplémentaire dans la stratégie qui vise à mettre le *smartphone* au centre de tous les usages, y compris pour des usages jusque là réservés aux composants sécurisés (les tentatives de paiement via le téléphone, et les « e-billets » vont dans ce sens).

Un second facteur qui pousse à faire disparaître l'objet carte à puce est la possibilité de lever les limitations de calcul et de mémoire liées à la carte à puce, et ainsi permettre de nouveaux usages. Par exemple, la télévision à péage, dont le marché est en plein essor avec les chaînes payantes de la Télévision Numérique Terrestre (TNT), les « bouquets télévisuels » offerts par les fournisseurs d'accès à internet [2], et le dématérialisé dans son ensemble (*Video On Demand*), chiffre traditionnellement non pas ses contenus, mais la clé d'accès aux contenus. En effet, le déchiffrement des contenus audiovisuels en

2. et utilisés comme des méthodes de différentiation dans les stratégies marketing

temps réel par une carte à puce n'est pas réalisable techniquement avec l'équipement actuel. Un principe prometteur dans ce domaine de recherche serait donc de déplacer la responsabilité du déchiffrement, traditionnellement attribuée à la carte à puce, vers une zone intégrée au processeur principal de la plateforme appelée « TEE ». Les TEE ont pour objectif de fournir plus de sécurité que le reste du processeur principal, tout en profitant de l'essentiel de ses performances, permettant par exemple de déchiffrer du contenu audiovisuel à la volée. Aujourd'hui les TEE ne fournissent néanmoins pas le niveau de sécurité d'un composant sécurisé [3], et des approches hybrides sont préconisées. Par exemple, la carte à puce peut utiliser sa clé de chiffrement maître (bien protégée par la carte) pour construire des sous-clés de chiffrement de faible durée de vie, servant au déchiffrement par le TEE. Ainsi la compromission d'une sous-clé de chiffrement ne permet de déchiffrer le contenu que pendant quelques heures, ce qui limite grandement l'intérêt de l'attaque.

Troisième facteur motivant la recherche d'un remplaçant à la carte à puce, l'arrivée prochaine de « l'Internet des Objets ». Dans un contexte qui se veut ouvert à de très nombreux acteurs de type *startup*, on imagine l'essor des *Components Off the Shelf* ou "composants sur l'étagère" (COTS) [4], qui ne sont dans leur état actuel que peu compatibles avec le processus de certification de composant sécurisé. Les solutions de type Java Card diminuent le coût de développement, mais on est encore loin d'une *carte blanche* dont la garantie de sécurité serait totale, mais qui serait programmable par tout un chacun.

Face à ce constat, qui est celui du temps qui passe—à quarante ans, la carte à puce a déjà fait preuve d'une longévité exceptionnelle dans le domaine de l'informatique—la table ronde s'est voulue rassurante. Le besoin de composants sécurisés n'est pas près de disparaître, et si le format « carte à puce » est appelé à évoluer, l'expertise et la recherche associées à la sécurité resteront nécessaires.

1.1.3. Processus de certification d'un composant sécurisé

Le processus de certification vise à évaluer de façon normalisée les revendications de sécurité faites par les fabricants d'un composant sécurisé. La certification permet aux clients de produits sécurisés de les choisir en connaissant leur niveau de sécurité, et garantit en principe l'équivalence en termes de sécurité entre deux produits de même niveau de sécurité. Le processus de certification garantit également que les guides d'utilisation des composants sécurisés précisent toutes les conditions à respecter par le client et l'utilisateur du système pour atteindre le niveau de sécurité du certificat émis.

Le processus de certification est établi dans le cadre des Critères Communs (CC) [5], qui constituent la norme ISO 15408 et permettent aux certificats émis par n'importe quelle agence nationale d'être valables à l'international, et notamment en Europe, aux État-Unis et au Canada [6]. Les CC sont le produit de la mise en commun de plusieurs

3. Qualcomm nous fournit un exemple récent de vulnérabilité liée aux TEE : http://www.silicon.fr/il-est-possible-de-casser-le-chiffrement-des-smartphones-android-152038.html.

4. expression désignant des composants logiciels et matériels disponibles sur le marché, combinés et réutilisés tels quels par l'entreprise, par opposition à des composants personnalisés conçus en interne par l'entreprise. Dans le milieu purement logiciel, on parle aussi de *middleware*.

5. Disponibles en intégralité sur http://www.commoncriteriaportal.org/

6. La liste plus précise des pays de l'agrément est disponible à l'adresse https://www.

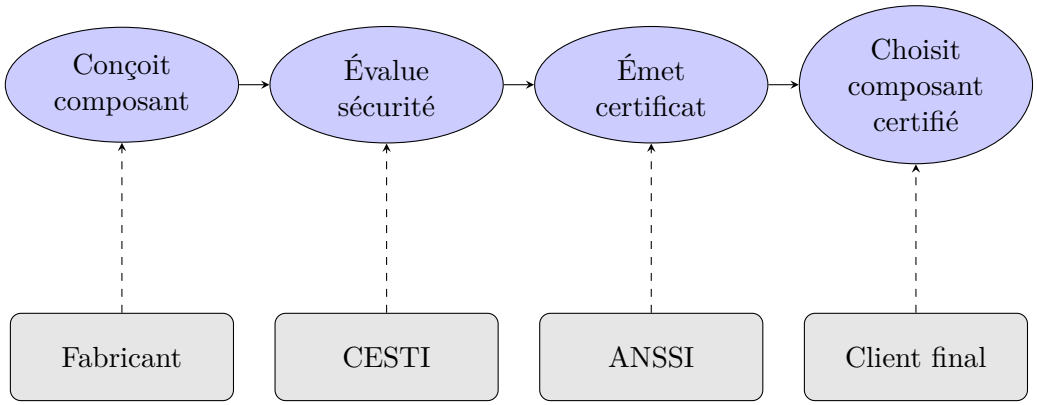


FIGURE 1.4. – Les acteurs et leur rôle dans le schéma de certification français

normes nationales et internationales préexistantes dans le *Common Criteria Recognition Arrangement* passé en 1999. Un processus plus léger que la certification CC, la Certification Sécuritaire de Premier Niveau (CSPN) [7], a été mis en place par l'Agence Nationale pour la Sécurité des Systèmes d'Information (ANSSI) en 2008.

1.1.3.1. Les acteurs

Les acteurs d'une certification de produit sont : un fabricant, un laboratoire d'évaluation, une agence nationale de certification et un client final. Ils sont illustrés sur la figure 1.4, et on détaille leur rôle ci-dessous :

- **Le fabricant** a conçu un produit qu'il souhaite vendre comme sécurisé. Il rédige une demande d'évaluation à l'agence nationale de certification. Une fois la demande acceptée, il se met en relation avec un laboratoire d'évaluation.
- **Le Centre d'Évaluation de la Sécurité des Technologies de l'Information (CESTI) est un laboratoire d'évaluation** agréé par l'agence nationale de certification, qui effectue l'évaluation puis produit un rapport technique d'évaluation.
- **L'agence nationale de certification** reçoit le rapport technique d'évaluation, le valide et émet le certificat. En France, c'est l'ANSSI qui joue ce rôle, en Allemagne le Bundesamt für Sicherheit in der Informationstechnik (BSI).
- **Le client final** est intéressé par le produit et ses propriétés de sécurité (en lien avec son propre besoin en sécurité et les normes en vigueur). Il l'achète, et utilise le guide du composant sécurisé pour connaître les règles d'utilisation du produit.

1.1.3.2. Vision globale du processus

Les CC fournissent un cadre au processus d'évaluation, via plusieurs étapes délimitées : la description de la cible de sécurité, les engagements du fabricant pour contrer les menaces de sécurité, le choix d'un niveau d'assurance, et enfin les tâches d'évaluation pour le fabricant et surtout le CESTI.

commoncriteriaportal.org/ccra/members/

7. Voir http://www.ssi.gouv.fr/administration/produits-certifies/cspn/

EAL	Description
EAL1	Functionally tested
EAL2	Structurally tested
EAL3	Methodically tested and checked
EAL4	Methodically designed, tested, and reviewed
EAL5	Semiformally designed and tested
EAL6	Semiformally verified design and tested
EAL7	Formally verified design and tested

Table 1.1. – Niveaux d'assurance d'évaluation

Cible de sécurité. Le fabricant doit rédiger un document qui décrit la *Security Target* ou "cible de sécurité" (ST) qu'il vise pour la certification. Ce document commence par la description du composant sécurisé, qu'on appelle la *Target Of Evaluation* ou "cible de l'évaluation" (TOE). Cette description permet à l'évaluateur de connaître les caractéristiques du composant, et de pouvoir rechercher les réponses aux questions techniques qu'il ne manquera pas de se poser pendant l'évaluation.

Menaces et objectifs de sécurité. Ensuite, le fabricant décrit le problème de sécurité, c'est-à-dire l'ensemble des menaces potentielles pour le composant sécurisé. Ceci permet de fixer ce qui doit être protégé. Concrètement, après la description en langage naturel des menaces, le fabricant énumère des objectifs de sécurité qui doivent répondre à toutes les menaces. Pour couvrir les objectifs, le fabricant choisit ensuite des *Security Functional Requirement* ou "exigence fonctionnelle de sécurité" (SFR) parmi la liste disponible dans le second tome des CC [CCR12a]. Chaque SFR décrit un certain nombre de fonctionnalités à intégrer par le fabricant dans la conception de son composant. Dans la pratique, le choix des SFR n'est pas libre, mais dicté par le type de composant sécurisé. Pour les menaces et composants courants, il existe même des *Protection Profiles* ou « profils de protection » (PPs) préétablis, qui prédéfinissent le problème de sécurité (dont les SFR à choisir). Ainsi, pour pouvoir prétendre à l'appellation de TEE, le fabricant doit suivre le PP correspondant [8].

Choix du niveau d'assurance. Le choix de l'*Evaluation Assurance Level* ou "niveau d'assurance d'évaluation" (EAL) détermine la quantité et la complexité des tâches à effectuer par l'évaluateur, et spécifie donc à quel point le composant a été testé. Ce choix n'est pas anodin, car le niveau d'assurance détermine quels marchés sont ouverts au composant sécurisé. Différentes normes spécifient ainsi un EAL minimum pour les satisfaire. Les EAL sont articulés en sept niveaux croissants de 1 à 7, où chaque niveau a des exigences de vérification plus élevés, donc un coût plus élevé, mais une assurance de sécurité plus élevée. Les différents EAL sont détaillés dans le tome III des CC [CCR12b]. Le tableau 1.1 résume l'objectif donné dans les CC à chaque EAL.

8. TEE Protection Profile. Tech. Rep. GPD SPE 021. GlobalPlatform, november 2014, http://www.globalplatform.org/specificationsdevice.asp

Assurance Class Name	Description
ASE	Security Target evaluation
AGD	Guidance documents
ADV	Development
ATE	Tests
ALC	Life-cycle support
AVA	Vulnerability assessment

TABLE 1.2. – Classes de SAR

Exigences d'évaluation L'évaluation est découpée en classes de *Security Assurance Requirement* ou "exigence d'assurance de sécurité" (SAR). Chaque SAR représente une exigence pour le fabricant et un ensemble de tâches pour l'évaluateur. Pour chaque SAR, il y a des niveaux croissants de sécurité (par exemple, `AVA_VAN5` est plus élevé que `AVA_VAN4`). Le niveau requis de la SAR est déterminé par l'EAL choisi, sachant que les EAL peuvent être « augmentés » par d'autres SAR (comme par exemple `EAL4+ALC_DVS2`, qui désigne « toutes les SAR requises par EAL4 **et** la SAR `ALC_DVS` de niveau 2 »). À chaque niveau est associé un ensemble de tâches décrites dans le document compagnon des CC, la *Common methodology for information technology security evaluation* (CEM). Les SAR sont regroupées par classes, le tableau 1.2 décrit les classes de SAR définies dans le tome III des CC [CCR12b]. La classe ASE correspond à l'évaluation du document qui décrit la cible de sécurité, AGD la vérification que le guide du composant est complet, ADV la vérification que les spécifications, conceptions et implémentations des fonctions de sécurité sont bien complètes et couvrent toutes les SFR, ATE la vérification des tests effectués par le fabricant sur son composant sécurisé et des tests indépendants effectués par l'évaluateur, ALC la vérification des pratiques de cycle de vie du composant (dans le cas d'une carte à puce, on vérifie notamment les phases de fabrication et de personnalisation) et enfin AVA recoupe l'analyse des vulnérabilités du produit et les tests de résistance aux attaques.

1.1.3.3. Cas de la carte à puce

La figure 1.5 instancie le schéma de certification dans le cas particulier de la carte à puce, où les éléments matériels sont conçus par des fondeurs spécialisés dans le matériel sécurisé, tandis que les applications de la carte sont implémentées par des développeurs spécialisés dans le logiciel embarqué. Dans ce cas, le développeur est lui-même fabricant de composants sécurisés, et utilise un composant déjà certifié (la carte du fondeur) comme support de son propre produit. Ce type de produit, dit « en composition », donne lieu successivement à une évaluation pour la carte et une évaluation pour le logiciel embarqué. Ces évaluations peuvent être menées par des CESTI différents.

Les cartes à puce sont typiquement protégées au niveau `EAL4+AVA_VAN_5+ALC_DVS2`. Les cartes à vocation de devenir passeports électroniques sont `EAL5+AVA_VAN5+ALC_DVS2`. Dans le cas des cartes bancaires, la carte doit également respecter la norme EMVCo [9]

9. Schéma de certification privé de cartes bancaires qui regoupe notamment Europay, Mastercard

et être acceptée par le GIE-carte bancaire [10]. La classe de SAR AVA joue donc un rôle important dans la protection des cartes à puce. On la détaille dans la section suivante.

et VISA.

10. Groupement inter-entreprises qui sélectionne individuellement les cartes bancaires pouvant être achetées par les différentes banques françaises.

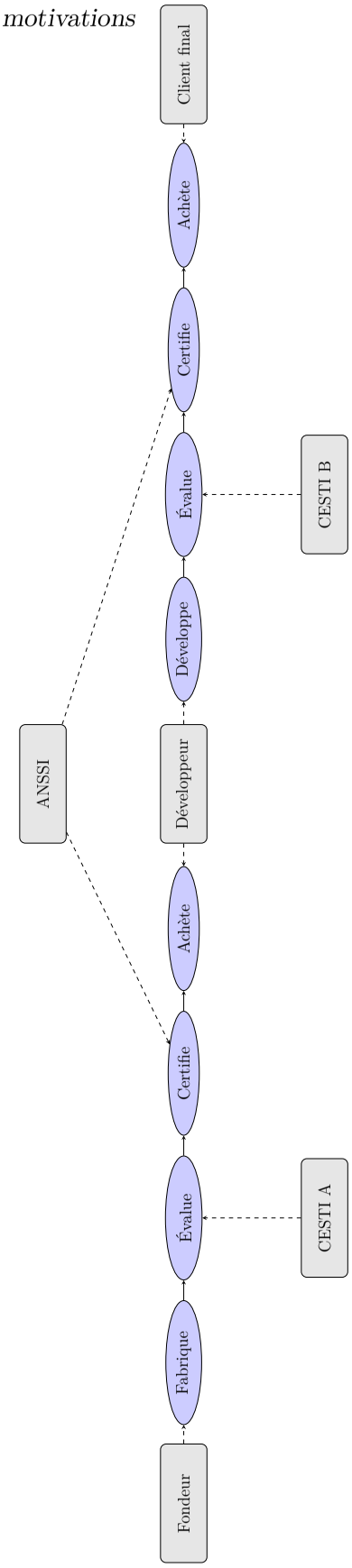


FIGURE 1.5. – Le schéma de certification appliqué au cas de la carte à puce

1.1.3.4. La classe AVA

Parmi les classes de SAR, la classe AVA est un peu particulière en ce qu'elle ne regroupe en fait qu'une seule SAR : `AVA_VAN`. Le but de cette SAR est de faire le lien entre la conformité (c'est-à-dire la documentation de la TOE) et l'efficacité (les tests de sécurité). Dans cette tâche, la majeure partie du travail est fournie par l'évaluateur. Elle requiert qu'il ait en sa possession la TOE, et dans un cadre de composant avec du logiciel, le code du logiciel embarqué dans le composant. L'évaluateur établit un plan de test, puis le déroule et écrit un rapport sur les résultats obtenus. Dans un contexte carte à puce, l'évaluation comprend systématiquement `AVA_VAN5`, qui correspond au niveau le plus élevé. L'évaluateur doit donc construire le plan de test le plus complet possible.

Pour définir le plan de test, l'évaluateur utilise deux vecteurs :

— Les attaques connues sur des composants similaires, c'est-à-dire les attaques faisant partie de l'état de l'art. L'état de l'art est défini en pratique par ce qu'on trouve dans les documents produits par les groupes de travail dédiés au type de composant sécurisé considéré. Pour la carte à puce, c'est le JHAS qui produit l'*Application of Attack Potential to Smartcards* de la *Joint Interpretation Library* (JIL) [Lib13], détaillant les attaques à l'état de l'art et leur cotation. Si les circonstances l'imposent (attaque très facile à mettre en œuvre, ou avec un impact *a priori* très important), les évaluateurs doivent également tenir compte des attaques publiées dans la littérature avant leur inclusion dans la JIL.

— L'analyse de code pour la recherche de vulnérabilités. L'évaluateur examine le code de l'application et détecte des vulnérabilités potentielles, qui donnent lieu à des tests supplémentaires dans le plan de test pour essayer de les mettre en évidence.

Après avoir déroulé le plan de test, l'évaluateur classe les résultats de chaque attaque testée. Pour cela, l'évaluateur utilise la notion d'attaque réussie.

Définition 1.1 *Attaque réussie*
Attaque qui compromet une ou plusieurs des propriétés de sécurité du composant sécurisé visé.

Si l'attaque a échoué, aucune suite n'est donnée. Si l'attaque a réussi, il faut établir sa cotation. Pour ce faire, on utilise une table, donnée dans le guide CEM, et spécialisée dans la JIL pour le cas des cartes à puce. Cette table, reproduite par le tableau 1.3, permet d'attribuer un score à différents facteurs dans deux catégories : d'une part l'identification de l'attaque (l'effort nécessaire pour détecter l'existence de l'attaque) et d'autre part son exploitation (l'effort nécessaire au succès de l'attaque une fois identifiée). Les scores attribués à chaque facteur sont sommés et le score total est comparé à un seuil qui détermine directement le niveau d'`AVA_VAN` auquel le composant peut prétendre. Par exemple, si une attaque réussie est cotée à un score inférieur ou égal à 31, alors le composant n'est pas `AVA_VAN5`. Dans le cas contraire, on considère que l'attaque ne met pas directement en jeu la sécurité du composant. Cependant, son existence est consignée en tant que « vulnérabilité résiduelle » dans le rapport technique d'évaluation (ETR pour *Evaluation Technical Report*) rendu par l'évaluateur à l'issue du processus

d'évaluation. Les surveillances annuelles du certificat émis devront alors vérifier que la cotation des vulnérabilités résiduelles n'a pas diminué.

On reviendra en détail sur la partie du processus de certification qui concerne la robustesse à l'injection de fautes dans la section 1.3.3. On gardera alors à l'esprit qu'il ne s'agit là que d'un petit sous-ensemble du processus complet.

Factors	Identification	Exploitation
Elapsed Time		
< one hour	0	0
< one day	1	3
< one week	2	4
< one month	3	6
> one month	5	8
Expertise		
Layman	0	0
Proficient	2	2
Expert	5	4
Multiple Expert	7	6
Knowledge of the TOE		
Public	0	0
Restricted	2	2
Sensitive	4	3
Critical	6	5
Very critical hardware design	9	NA
Access to TOE		
< 10 samples	0	0
< 30 samples	1	2
< 100 samples	2	4
> 100 samples	3	6
Equipment		
None	0	0
Standard	1	2
Specialized	3	4
Bespoke	5	6
Multiple Bespoke	7	8
Open samples		
Public	0	NA
Restricted	2	NA
Sensitive	4	NA
Critical	6	NA

TABLE 1.3. – Facteurs de l'*Attack Potential for smartcards* tirés de la JIL [Lib13]

1.2. Attaques par perturbation

Le tableau 1.4 regroupe les différents types d'attaques sur composants sécurisés. Les attaques sont divisées en deux catégories, suivant qu'il s'agit d'attaques matérielles ou d'attaques logicielles. Les attaques logicielles exploitent exclusivement des vulnérabilités logicielles, c'est-à-dire causées par la façon dont le code du composant est écrit. Les attaques par canaux cachés sont des attaques passives qui sont basées sur l'observation des traces émises par le composant (par exemple, les traces de consommation). Les attaques invasives, à l'inverse, sont des attaques qui modifient de façon permanente le composant attaqué, par exemple pour le décapsuler ou comprendre sa conception. Le lecteur curieux de ces autres types d'attaques pourra se référer à l'annexe A de ce manuscrit pour un rapide état de l'art. Dans cette section, on détaille les attaques par perturbation.

Attaques logicielles	Attaques matérielles		
	Canaux cachés	Invasif	Perturbation

TABLE 1.4. – Vecteurs d'attaque sur composants sécurisés

1.2.1. Définition

Les attaques par perturbation consistent à modifier un ou plusieurs paramètres physiques de l'environnement afin de perturber le fonctionnement normal du composant sécurisé. Il s'agit d'attaques semi-invasives, c'est-à-dire qu'elles ont la caractéristique de modifier le matériel de façon non-permanente.

Les attaques par perturbation sont évoquées dès 1996 par Voas dans [Voa96], mais c'est 1997 qui marque le point de départ officiel du domaine, avec l'article de Boneh *et al.* [BDL97] où les auteurs démontrent qu'une unique perturbation durant un calcul cryptographique permet de retrouver une clé privée RSA et de casser d'autres algorithmes cryptographiques. L'attaque exploite une propriété mathématique du théorème des restes chinois, traditionnellement utilisé pour accélérer les calculs dans les algorithmes cryptographiques basés sur le problème de la factorisation. On parle de l'attaque BellCore, du nom du laboratoire de Boneh, DeMillo et Lipton. Les auteurs proposent également une contre-mesure à l'attaque, sous la forme d'une vérification d'intégrité du calcul cryptographique. Si la vérification détecte une incohérence, le résultat du calcul n'est pas fourni à l'utilisateur, empêchant de factoriser la clé.

En 2007, Kim et Quistater proposent de nouvelles attaques sur RSA qui ont la particularité d'utiliser plusieurs perturbations durant une seule exécution [KQ07]. Cette possibilité d'attaque par perturbations multiples ajoute un nouveau niveau de complexité, puisque les attaques peuvent alors perturber à la fois le calcul et les contre-mesures qui visent à détecter les fautes provoquées par une perturbation.

La fin de cette section détaille l'équipement spécialisé actuellement utilisé pour effectuer de telles perturbations.

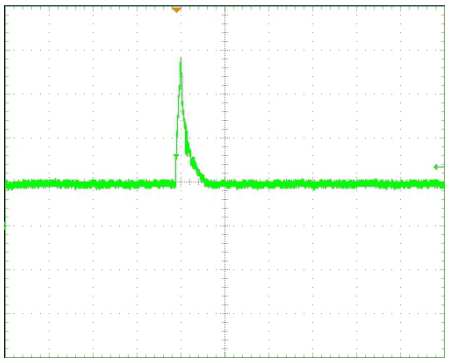


FIGURE 1.6. – Pic de consommation dû à un glitch

1.2.2. Équipement d'attaque

L'équipement d'attaque utilisé varie en fonction des protections du composant visé. L'équipement est caractérisé par son coût, les opérations de préparation nécessaires pour pouvoir l'utiliser avec le composant (par exemple, décapsulation d'une carte à puce pour utiliser un laser), l'existence de contre-mesures répandues et efficaces contre l'équipement, et enfin le contrôle spatio-temporel qu'il donne à l'attaquant. Le contrôle spatio-temporel désigne la précision avec laquelle un attaquant peut cibler son attaque suivant les deux dimensions de paramètres : temporels et spatiaux. Les paramètres temporels d'une attaque recouvrent l'instant précis de déclenchement de la perturbation ainsi que sa durée. Les paramètres spatiaux recouvrent la possibilité de cibler une partie spécifique du composant sécurisé, par exemple les registres ou la mémoire. Un bon contrôle permet de fixer avec précision ces paramètres, tandis qu'un mauvais contrôle réduit la plage ou la reproductibilité des valeurs accessibles.

1.2.2.1. Attaque par glitch

Les attaques par glitch, appliquées aux attaques par perturbation en 1998 par Anderson et Kuhn [AK98], reposent sur une impulsion électrique brutale et importante envoyée sur l'alimentation ou la terre du composant.

Le contrôle temporel est bon pour l'application du glitch en elle-même, mais mauvais pour la durée du glitch, qui dépend du circuit considéré. Le contrôle spatial est presque inexistant, les deux seuls points d'accès étant l'alimentation et la terre du composant.

Les avantages de cette méthode sont son coût très réduit (un générateur d'impulsion électrique) et qu'elle ne requiert pas d'accès au silicium (inutile d'ouvrir le composant via une attaque invasive). Les contre-mesures matérielles pour ce type d'attaque sont extrêmement efficaces, et les composants sécurisés certifiés `AVA_VAN5` doivent tous résister aux glitches, ce qui les rend généralement impraticables sur des composants modernes.

1.2.2.2. Attaque par lumière focalisée

En 2003, Skorobogatov et Anderson annoncent un nouveau type d'attaque : l'illumination très localisée d'un transistor par un laser lui fait conduire le courant, ce qui

FIGURE 1.7. – Un laser

perturbe son fonctionnement [SA03]. Les auteurs utilisent cette méthode pour effectuer l'attaque BellCore.

Aujourd'hui encore, le laser est un équipement de choix pour les attaquants, avec un excellent contrôle temporel et spatial (la lumière étant focalisée). Les inconvénients de la méthode sont son coût relativement élevé par rapport aux autres méthodes [11], et la nécessité d'ouvrir le composant pour accéder au silicium, ce qui requiert une attaque invasive.

Diverses contre-mesures matérielles telles que des détecteurs de lumière ont été proposées, mais ces contre-mesures sont coûteuses, et ne protègent pas entièrement le composant, en plus d'augmenter sensiblement le risque de faux positifs.

1.2.2.3. Attaque par champ électromagnétique

Durant les essais nucléaires de la Seconde Guerre Mondiale, il fut constaté que les impulsions électromagnétiques causées par les détonations de bombes H peuvent détruire complètement des appareils électroniques. Dès 2002, le champ magnétique créé par une bobine d'une sonde électromagnétique soumise à un fort voltage est testé par Quisquater et Sadyme [QS02] pour perturber des calculs, mais il faut attendre 2007 pour que ce principe soit appliqué par Schmidt et Hutter pour effectuer l'attaque BellCore à l'aide d'un générateur d'impulsions électromagnétiques [SH07]. Les attaques par champ électromagnétique ont gagné en popularité ces dernières années, avec notamment les expériences menées par Ordas *et al.* [Ord+15] en 2014, et la démocratisation de l'équipement d'attaque, avec l'apparition de COTS dédiés à l'injection EM (initialement utilisés industriellement pour préserver les aliments [Cam+08]).

L'avantage théorique principal de l'injection électromagnétique sur le laser est la possibilité d'attaquer sans ouvrir le composant, et à distance. Dans la pratique, sur des composants sécurisés, l'ouverture se révèle généralement nécessaire. Le contrôle temporel des attaques électromagnétique est un peu moins fin que celui des glitches et du laser, car il faut charger le générateur d'impulsions pendant un certain temps avant

11. Il serait de l'ordre de 150k€ en 2015 d'après [BJ15].

Équipement	Temporel	Spatial	Contre-mesures	Ouverture	Coût
Glitch	Médiocre	Mauvais	Efficaces	Non	Faible
Laser	Excellent	Excellent	Peu Efficaces	Oui	Élevé
EM	Médiocre	Bon	Peu Efficaces	Oui	Modéré

TABLE 1.5. – Récapitulatif des performances des différents équipements

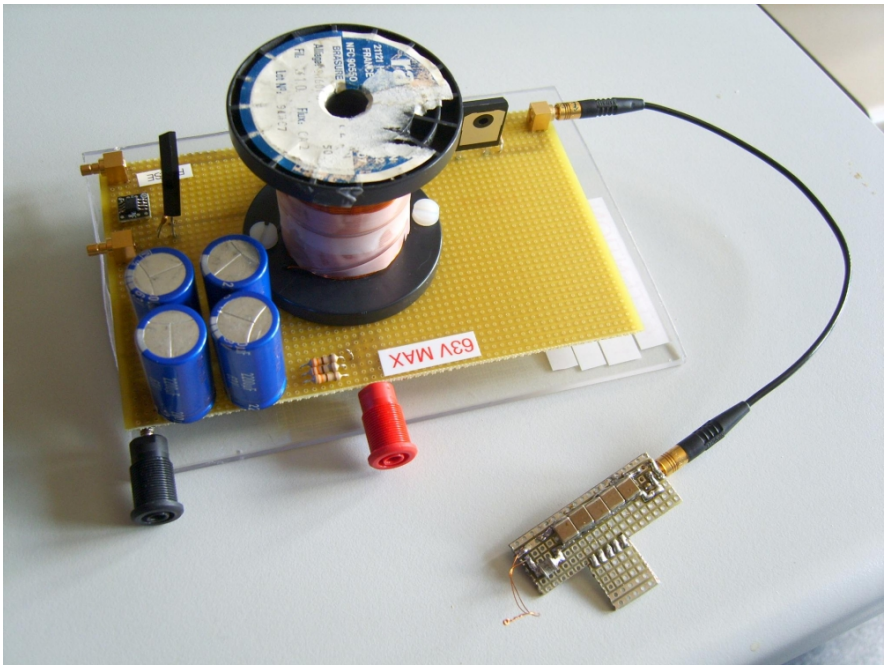


FIGURE 1.8. – Un équipement d'attaque EM

de déclencher l'attaque. Le contrôle spatial est bien meilleur que celui des attaques par glitches, et est de l'ordre de celui du laser, même si dans la pratique il reste moins focalisé.

1.2.2.4. Récapitulatif

Le tableau 1.5 résume les différents caractéristiques des équipements présentés dans la section.

1.2.3. Quelques attaques

1.2.3.1. Dépassement induit de tampons

Les vulnérabilités de dépassement de tampons (*buffer overflow*) sont un type de vulnérabilité bien connu en sécurité logicielle, qui a causé le développement de nombreuses analyses pour sa détection au tournant des années 2000 [Cow+98 ; LE01 ; Cow+03]. Le dépassement de tampons se produit lorsqu'un programme accède à un indice d'un

tableau situé en dehors des bornes du tableau (après, ou avant). Ce type de vulnérabilité a des conséquences très graves sur la sécurité du programme, voire du système qui exécute le programme : accéder à une case mémoire qui n'est pas dans le tableau permet de lire ou écrire un emplacement mémoire non prévu par le programme, ce qui peut amener à la corruption de secrets ou dans le pire des cas à la possibilité pour l'attaquant d'exécuter du code arbitraire.

Dans le cadre des composants sécurisés, les vulnérabilités de dépassement de tampons sont soigneusement vérifiées pendant l'évaluation par un CESTI, aidé par des outils matures, si bien qu'il est très peu probable d'en trouver dans un produit certifié. Néanmoins, en utilisant les attaques par perturbation, Fouque *et al.* parviennent en 2012 à obtenir des effets similaires à ceux des vulnérabilités par dépassement de tampons [FLV12].

1.2.3.2. Applets malicieuses sous Java Card

Java Card est un environnement d'exécution pour un langage dérivé de Java et spécialisé dans les applications pour cartes à puce. Une carte Java Card se veut un environnement plus ouvert qu'une carte à puce standard, avec la possibilité de déployer plusieurs applications dans la carte pendant son cycle de vie (y compris potentiellement après sa délivrance au porteur), tout en assurant pour chaque application un niveau de sécurité et d'isolation compatibles avec le statut de composant sécurisé.

Cette relative ouverture, unique pour les composants sécurisés, ainsi que le langage Java qui fournit une plate-forme unique de développement sur un large éventail de composants, ont donné lieu à un intérêt particulier dans le domaine des attaques, et en conséquence à une littérature assez fournie sur le sujet [MP08 ; IL09 ; SIL11 ; LB15 ; LR15]. Dans cette littérature, les attaques par perturbation sur Java Card jouent un rôle de choix. On les utilise pour effectuer des attaques hybrides, c'est-à-dire pour provoquer des vulnérabilités logicielles à l'aide des attaques matérielles [BTG10 ; BIL11].

1.3. Injection de fautes

En test logiciel, une faute est une différence dans le code d'un programme par rapport au code attendu. Certaines fautes résultent en une erreur, c'est-à-dire une différence visible dans le comportement ou les sorties du programme, et certaines erreurs sont des vulnérabilités, c'est-à-dire des faiblesses exploitables par un attaquant pour compromettre des propriétés de sécurité assurées par le programme.

1.3.1. Injection de fautes et attaque par perturbation

Les conséquences des attaques par perturbation présentées dans la section 1.2 peuvent se modéliser par des injections de fautes dans le code exécuté par le composant sécurisé. Les fautes ont la particularité d'être *injectées* dans le code, ce qui signifie que, contrairement au test classique, elles ne sont pas initialement présentes dans le code évalué. Cette particularité rend contre-intuitive l'évaluation du code, puisque l'évaluateur doit imaginer la présence de fautes et déterminer leurs conséquences au lieu de simplement les détecter. Cette tâche est rendue particulièrement ardue par le fait qu'on ne sait

pas décrire avec certitude quelles fautes seront injectées, et donc quelles fautes sont à considérer pendant l'évaluation.

1.3.2. Injection de fautes et code

Depuis l'attaque BellCore sur RSA-CRT, l'histoire des attaques par perturbation s'est construite avec un lien très fort à la cryptographie : beaucoup de travaux étudient les attaques par fautes spécifiquement dans ce cadre [Bar+14 ; Chr+13 ; Chr13 ; RG14b ; RG14a ; BOS03 ; TK10 ; KQ07]. Le cadre cryptographique, quoique important, n'est qu'une partie de la problématique liée à l'injection de fautes, et il nous a paru important de considérer un cadre plus général au cours de la thèse. L'injection de fautes est un problème fondamental qui touche à la nature du code.

1.3.2.1. L'exemple `VerifyPIN`

Tout au long de ce manuscrit, on fera référence à l'exemple dit `VerifyPIN`. L'exemple `VerifyPIN` est une fonction de vérification de code PIN (*Personal Identification Number*) permettant l'authentification auprès de la carte. Si le code entré par l'utilisateur correspond au code PIN de référence enregistré dans la carte, l'utilisateur est authentifié, ce qui lui donne accès à davantage de fonctionnalités de la carte. Les codes PIN sont utilisés dans de nombreuses applications, les plus évidentes étant la carte bleue, où la majorité des transactions nécessite le code PIN de l'utilisateur, et la carte SIM, qui nécessite un code PIN pour donner accès aux fonctions liées à l'abonnement de l'utilisateur.

`VerifyPIN` est notre exemple de référence, de par sa simplicité, son importance (les fonctions de vérification de PIN sont un enjeu sécuritaire), et l'absence de propriétés particulières dues à un algorithme cryptographique.

Le listing 1.1 donne une implémentation du `VerifyPIN`. Le code PIN entré par l'utilisateur est contenu dans la variable `g_userPin` et comparé au code PIN de référence `g_cardPin` pour fixer le jeton d'authentification `g_authenticated` à la valeur `BOOL_TRUE` ou `BOOL_FALSE`.

Une attaque peut-être considérée réussie si l'attaquant est authentifié suite à l'attaque alors que le code PIN entré est différent du code PIN de référence [12]. Dans le listing 1.1, ces conditions signifient que la variable `g_authenticated` vaut `BOOL_TRUE`, même si la variable `g_userPin` contient des valeurs différentes de `g_cardPin`.

1.3.2.2. Notion de modèle de faute

Comme expliqué en 1.3.1, les fautes injectées lors d'une attaque par perturbation doivent être prises en compte lors de l'évaluation du code. Pour décrire ces fautes, on utilise la notion de modèles de faute.

12. D'autres scénarios existent, par exemple, tester une valeur de PIN sans décrémenter le compteur d'essais, afin de pouvoir effectuer une attaque exhaustive.

</>Code source

```
1  extern SBYTE g_ptc;
2  extern BOOL g_authenticated;
3  extern UBYTE g_userPin[PIN_SIZE];
4  extern UBYTE g_cardPin[PIN_SIZE];
5  void verifyPIN() {
6    int i;
7    BOOL status, diff;
8    g_authenticated = BOOL_FALSE;
9    if (g_ptc >= 0) {
10     g_ptc--;
11     diff = BOOL_FALSE;
12     for (i = 0; i < PIN_SIZE; ++i) {
13       if (g_userPin[i] != g_cardPin[i]) {
14         diff = BOOL_TRUE;
15       }
16     }
17     if (diff == BOOL_FALSE) {
18       g_ptc = 3;
19       g_authenticated = BOOL_TRUE; // Authentication
20     }
21   }
22 }
```

Listing 1.1. – Implémentation naïve de `VerifyPIN`

Définition 1.2 *Modèle de faute*
Ensemble de règles de construction des fautes admises lors de l'analyse.

Les modèles de faute reflètent ainsi les pouvoirs de l'attaquant, c'est-à-dire l'ensemble des transformations qu'il est en mesure d'effectuer sur le code.

Le listing 1.2 présente un exemple d'implémentation fautée. Dans cette implémentation, la comparaison `i < PIN_SIZE` (ligne 6) a été inversée en `i >= PIN_SIZE` par rapport à l'implémentation d'origine du listing 1.1. Cette faute a pour conséquence qu'aucune itération de la boucle n'est exécutée, et l'exécution termine donc avec le jeton d'authentification à `BOOL_TRUE`, ce qui octroie l'authentification à l'attaquant. Cette faute peut s'interpréter comme une instance du modèle de faute de l'inversion de test, modèle qui regroupe toutes les fautes consistant en l'inversion d'un test. La littérature regorge de modèles de fautes [13], suivant l'emplacement fauté (mémoire, registre, opération arithmétique ou logique), que les fautes soient permanentes (modifient la valeur comme si la valeur fautée était écrite dans l'emplacement fauté, comme dans l'exemple) ou volatiles (modifient la valeur comme si la valeur fautée était lue une seule fois à la place de la valeur normalement contenue dans l'emplacement fauté, contrairement à

13. On y revient au chapitre 2.

</>Code source

```
1  void verifyPIN() {
2    // [...]
3    if (g_ptc >= 0) {
4      g_ptc--;
5      diff = BOOL_FALSE;
6      for (i = 0; i >= PIN_SIZE; ++i) {
7        if (g_userPin[i] != g_cardPin[i]) {
8          diff = BOOL_TRUE;
9        }
10     }
11     if (diff == BOOL_FALSE) {
12       status = BOOL_TRUE;
13       g_ptc = 3;
14       g_authenticated = BOOL_TRUE; // Authentication
15     }
16   }
17 }
```

Listing 1.2. – Implémentation fautée de `VerifyPIN`

l'exemple), et qu'ils servent à décrire des fautes sur du code source, des graphes de flot de contrôle, des listings assembleur, ou du binaire.

1.3.2.3. Contre-mesures logicielles

Définition 1.3 *Contre-mesure logicielle*
Code spécifiquement conçu pour détecter l'injection de faute et réagir en conséquence.

Le spectre des réactions possibles de la carte s'étend de l'infection du résultat (transformation qui rend le résultat non significatif pour ne pas donner d'information à l'attaquant) à l'auto-destruction permanente de la carte, en passant par un mutisme de la carte (qui force à la redémarrer pour recommencer à l'utiliser).

Le listing 1.3 montre un exemple de contre-mesure, où le compteur de boucle est testé immédiatement après la sortie de boucle, et avant d'accepter l'authentification comme valide. Si une faute a eu lieu comme décrit dans le listing 1.2, la variable `i` ne vaut pas `PIN_SIZE` en sortie de boucle, et cette incohérence est détectée par la contre-mesure, ce qui permet à la carte de réagir (par exemple, en s'autodétruisant pour protéger ses secrets contre d'autres attaques qui pourraient réussir).

</>Code source

```
void verifyPIN() {
  // [...]
  if (g_ptc >= 0) {
    g_ptc--;
    diff = BOOL_FALSE;
    for (i = 0; i < PIN_SIZE; ++i) {
      if (g_userPin[i] != g_cardPin[i]) {
        diff = BOOL_TRUE;
      }
    }

    if(i != PIN_SIZE) {
      countermeasure();
    }

    if (diff == BOOL_FALSE) {
      g_ptc = 3;
      g_authenticated = BOOL_TRUE; // Authentication
    }
  }
}
```

Listing 1.3. – Implémentation avec contre-mesure de `VerifyPIN`

1.3.3. Processus d'évaluation de la robustesse à l'injection de fautes

Le processus d'évaluation de la robustesse à l'injection de fautes est une partie du processus d'évaluation de la sécurité des composants sécurisés présenté à la section 1.1.3, et en particulier une partie de la classe AVA. Son objectif est de répondre à la question « Le composant sécurisé est-il robuste à l'injection de fautes ? ». Cette question se décline en deux variantes suivant qu'on considère un composant destiné à accueillir du code avant l'ajout du code, où au contraire que la partie à évaluer est le logiciel embarqué sur le composant. Dans la suite de la thèse, on a choisi de s'intéresser spécifiquement au cas d'un logiciel embarqué. Comme annoncé à la section 1.1.3, la classe AVA est constituée d'une unique SAR, `AVA_VAN`, qui requiert de l'évaluateur la préparation et l'exécution d'un plan de test. En lien avec l'analyse de vulnérabilités logicielles classiques, et avec les pratiques au CESTI, on peut articuler ces phases suivant deux sous-processus :

— La *revue de code* (figure 1.9) vise à détecter des vulnérabilités à l'injection de fautes directement depuis le code source et le listing assembleur du logiciel testé. C'est une approche dite « boîte blanche », dans laquelle l'évaluateur utilise directement ses connaissances du système (notamment, le code) pour rechercher les vulnérabilités. Cette partie est notamment utilisée pour établir le plan de test.

— Les *tests de pénétration* (figure 1.10) visent à effectuer un grand nombre d'attaques physiques par perturbation, en utilisant l'équipement présenté à la section 1.2,

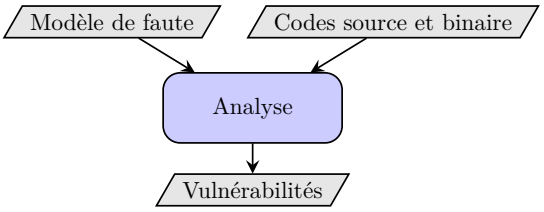


FIGURE 1.9. – Revue de code

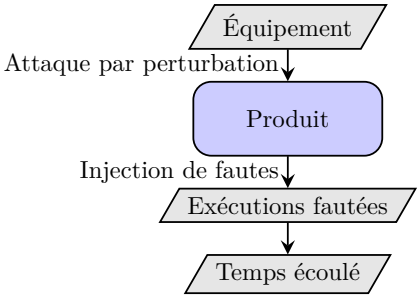


FIGURE 1.10. – Tests de pénétration

afin de rechercher des attaques réussies. C'est une approche « boîte noire » dans laquelle l'évaluateur met de côté ses connaissances du système pour le tester du point de vue d'un attaquant. Dans cette partie, les évaluateurs déroulent le plan de test.

Chaque attaque réussie découverte dans le cadre de ces deux sous-processus est cotée, comme décrit à la section 1.1.3, à l'aide du tableau 1.3, et en particulier le résultat des tests de pénétration est utilisé pour déterminer le facteur *elapsed time* (temps écoulé), qui correspond au temps nécessaire pour identifier, puis exploiter l'attaque.

Le processus d'analyse de robustesse à l'injection de fautes tel que présenté ouvre un certain nombre de problèmes, qui ont servi de base aux problématiques de recherche de la thèse.

1.4. Objectifs de la thèse

En conclusion de ce premier chapitre et en introduction des suivants, on présente les objectifs poursuivis pendant la thèse, et l'approche mise en œuvre à cette fin.

1.4.1. Améliorer le processus d'analyse de vulnérabilité à l'injection de fautes

L'objectif principal de la thèse a été l'amélioration et la rationalisation du processus d'analyse de vulnérabilités à l'injection de fautes. En effet le processus existant au début de la thèse, et tel qu'on l'a présenté en section 1.3.3, comprenait un certain nombre de défis majeurs à relever. Parmi ceux-ci, l'augmentation de la répétabilité du processus, et en particulier de la revue de code, a servi de point de départ à la thèse et a conduit à la réalisation d'un outil d'analyse de code pour la recherche de vulnérabilités à l'injection

de fautes.

La conception de cet outil a amené à considérer plusieurs problématiques de recherche, qu'on présente ci-dessous.

1.4.1.1. Comment modéliser les effets de l'injection de fautes sur le code ?

Les effets de l'injection de fautes sur le code sont encore mal compris, et loin d'être exhaustivement décrits. Il est très difficile pour un informaticien de raisonner avec des modèles d'exécution qui comportent des erreurs, où « deux et deux cessent de faire quatre, et se mettent à faire cinq ». Cette difficulté est exacerbée par le fait qu'on ne peut directement déterminer les fautes injectées dans du code : on ne peut qu'observer les différences fonctionnelles de comportement du composant introduites par des attaques par perturbation. En l'absence d'une modélisation fiable des effets de l'injection de fautes, l'évaluateur est contraint, soit de tester indépendamment chaque programme avec des attaques par perturbation sans pouvoir généraliser les résultats, soit d'utiliser un modèle de faute prédéfini dans la revue de code sans pouvoir en questionner la pertinence.

1.4.1.2. Comment faire passer à l'échelle l'analyse de vulnérabilité à l'injection de fautes ?

Par rapport au test logiciel traditionnel, on teste non pas un seul programme, mais un programme et toutes ses variantes. Les outils d'analyse de la littérature (qu'on détaille au chapitre 2) sont adaptés à l'analyse de petits exemples, en excluant les attaques par perturbations multiples. L'objectif qui découle de cette problématique a donc été d'étudier les possibilités d'analyse permettant le passage à l'échelle de programmes plus longs ou à des attaques par perturbations multiples.

1.4.1.3. Qu'est-ce qu'un code robuste à l'injection de faute ?

Idéalement, un code robuste à l'injection de faute se définit comme un code sur lequel aucune attaque par perturbation ne met en danger la sécurité. Cette posture théorique contraste fortement avec la réalité des évaluations, où ce type de garantie est très difficile à obtenir. Dans la pratique des CC, une attaque réussie ne met en jeu la robustesse du code que si sa cotation est trop basse par rapport au niveau de sécurité visé (voir section 1.1.3). La cotation des attaques joue donc un rôle central dans la détermination de la robustesse d'un code. Pourtant, dans le processus d'évaluation, seuls les tests de pénétration sont utilisés dans les cotations des vulnérabilités à l'injection de fautes. Un des objectifs de la thèse a donc été de définir des métriques calculables à partir des résultats de l'analyse de code, et permettant de quantifier la robustesse d'un code à moindre coût.

1.4.2. Construire un socle commun pour la recherche en attaques par perturbation

Au cours de la thèse, j'ai été confronté à un problème concernant les exemples de code disponibles pour l'injection de faute. En effet, le milieu des composants sécurisés

est un milieu de secrets industriels, et les implémentations sécurisées font partie du savoir-faire des développeurs. Par conséquent, elles ne sont pas publiques. De même, les différents outils d'analyse de vulnérabilité ne sont pas disponibles.

Cette situation conduit les chercheurs qui présentent leurs outils d'analyse à le faire sur des exemples artisanaux, souvent des exemples courts et peu représentatifs de l'injection de fautes dans son ensemble. De même, la vérification des contre-mesures présentées se fait avec des outils « maison », donc avec peu de cohésion et une faible reproductibilité.

Un des objectifs de la thèse a donc été de fournir un début de solution à ce problème, sous la forme d'un ensemble commun d'exemples conçus pour l'injection de fautes publiés en *open-source*. Cette construction s'est faite dans le cadre d'un projet DGA-ANR ASTRID, SERTIF [14], qui vise à rationaliser et automatiser autant que possible le processus d'analyse de robustesse contre l'injection de fautes, et dont l'une des principales contributions est cette collection de codes sécurisés. Le projet SERTIF regroupe un consortium constitué du laboratoire de recherche VERIMAG, de l'industriel développeur de composants sécurisés Safran Morpho, et du CESTI-LETI où j'ai fait ma thèse. Cette collaboration a permis d'assurer la représentativité des exemples de code proposés.

1.4.3. Une approche « de bout en bout »

Pour répondre à l'objectif d'amélioration du processus d'analyse de vulnérabilités à l'injection de faute, on propose dans le cadre de la thèse une nouvelle approche « de bout en bout », c'est-à-dire qui couvre tout le processus. L'approche proposée introduit une phase préparatoire conçue pour permettre l'inférence d'un modèle de faute adapté au composant attaqué. La seconde étape de cette approche est la simulation sous injection de faute, qui explore les conséquences du modèle de faute obtenu sur le code évalué. Enfin, on propose de nouvelles métriques quantitatives de robustesse, dont le calcul est basé sur les résultats des deux phases précédentes. La figure 1.11 résume notre approche.

Le chapitre 2 de ce manuscrit effectue un état de l'art des outils d'analyse de la littérature, le chapitre 3 détaille notre méthode d'inférence de modèle de faute. Le chapitre 4 présente le point de vue de l'évaluateur sur l'analyse de code, qui s'est traduit dans la thèse par la conception et l'implémentation d'un outil d'analyse de simulation d'injection de fautes. Le chapitre 5 discute l'extension des résultats du chapitre 4 au cas particulier de l'injection de fautes multiples indépendantes. Enfin le chapitre 6 clôt le manuscrit en faisant le bilan des travaux effectués et en présentant leurs perspectives.

14. Simulation pour l'Evaluation de la RobusTesse des applications embarquées contre l'Injection de Fautes, ANR-14-ASTR-0003-01, http://sertif-projet.forge.imag.fr/

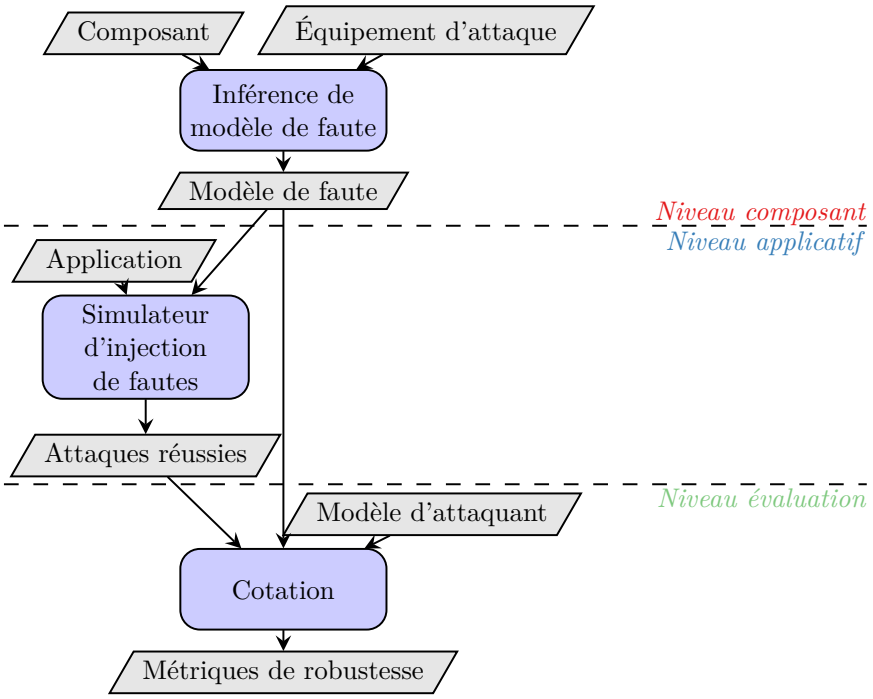


FIGURE 1.11. – Approche « de bout en bout »

Chapitre 2

Outils pour l'injection de fautes

> La liberté, c'est la liberté de dire que deux et deux font quatre. Lorsque cela est accordé, le reste suit.
>
> *(George Orwell,* 1984*)*

Résumé du chapitre

Dans ce chapitre, on s'intéresse aux différents outils existants pour simuler l'injection de fautes et ses effets. Ces outils ont été développés dans deux domaines de recherche différents, à savoir la tolérance aux fautes et l'analyse de vulnérabilités à l'injection de fautes. La tolérance aux fautes a largement étudié ces outils, quels apports a eu le domaine ? Quelles différences par rapport à l'injection de fautes ? Ce chapitre commence par un état de l'art des outils d'analyse de tolérance aux fautes, et étudie les liens entre ce domaine et celui de l'analyse de vulnérabilités. Dans une seconde partie, on liste les outils d'analyse de vulnérabilités existants, puis on tente de les comparer dans l'objectif de dégager des critères utiles de classification. En conclusion de ce chapitre, on offre quelques réflexions sur l'état actuel du domaine, puis sur les points communs et les différences inconciliables entre les outils d'analyse de vulnérabilités et ceux de tolérance aux fautes. Enfin, on ouvre sur les chapitres suivants en décrivant le besoin d'un outil portable, configurable en fonction du modèle de faute et pensé pour le passage à l'échelle des fautes multiples indépendantes.

Sommaire

2.1. Outils de tolérance aux fautes

Dans sa thèse de 2010, Piètre-Cambacédès étudie les liens qui unissent les domaines de la sûreté et de la sécurité [Piè10]. Les deux domaines de recherche ont en effet évolué séparément, alors que les problèmes résolus par chaque domaine sont d'une nature proche, qui appelle l'utilisation d'outils proches. En particulier, la sûreté bénéficie d'une histoire beaucoup plus longue, dans la continuité de la sûreté dans les autres domaines d'ingénierie (construction, avionique, trains), tandis que la sécurité s'est développée plus tardivement, mais en mettant davantage l'accent sur le volet informatique. L'injection de fautes s'inscrit exactement dans ce cadre. Comme on a pu le voir en section 1.2, les attaques par perturbation se développent à partir de 1996 ; par contraste, le problème des fautes d'origine naturelle s'est posé dès 1950 [SBN82, p. 155]. Cette différence temporelle a permis aux approches de simulation logicielle de fautes matérielles en tolérance aux fautes d'être popularisées dès 1988 [Seg+88], tandis que le premier outil en attaque par perturbation (à notre connaissance) est apparu en 2004 [Rot+04], soit 16 ans plus tard.

2.1.1. Parallèles et différences

2.1.1.1. Domaine d'application

On l'a vu, la sécurité s'intéresse aux composants sécurisés. Dans cette catégorie, on trouve notamment un certain nombre d'automates (distributeurs de billets/de tickets de transports en commun), les cartes à puce et leurs lecteurs. Tous ces composants visent un coût réduit. En sûreté, les systèmes qui se protègent contre les fautes sont des systèmes très sensibles, qui mettent en jeu la vie des personnes : voitures, avions, trains, centrales nucléaires, etc. De par leur importance critique, ces systèmes peuvent être très onéreux, les moyens de protection disponibles ne sont donc pas les mêmes, avec une majorité de protections logicielles dans la sécurité, et une majorité de protections matérielles dans la sûreté.

2.1.1.2. Nature des risques

Dans [PC10], Piètre *et al.* établissent la terminologie SEMA, qui vise à éclairer les différents sens qui ont pu être donnés aux mots *sécurité* et *sûreté* dans la littérature. À cette occasion, les auteurs relèvent comme différence majeure entre sécurité et sûreté la présence ou l'absence d'un adversaire malveillant. Cette différence de nature des risques induit une différence de gestion des risques. En sûreté, en absence de malveillance, on peut dire que le risque est *statique*, c'est-à-dire qu'il est connu et identifié clairement au moment de la conception du système, et restera le même durant la vie du système. En sécurité, à l'inverse, la présence d'un adversaire malveillant doté d'intelligence rend les

attaques imprévisibles, et la gestion des risques beaucoup plus difficile à conduire [1]. La sécurité peut donc se voir comme un jeu « du chat et de la souris », où les attaquants développent de nouvelles attaques et les défenseurs des contre-mesures en réaction. Ceci éclaire l'importance particulière, dans le processus de certification en sécurité, des documents qui regroupent les moyens d'attaques existants comme la CEM ou le JIL, qui servent de « jurisprudence » des attaques à contrer.

2.1.1.3. Répartition des fautes

En tolérance aux fautes, on cherche à se défendre contre un phénomène naturel de l'environnement (par exemple, le rayonnement cosmique, un milieu radioactif, le champ magnétique, etc.) ou dans le système lui-même (panne). On peut donc étudier ce phénomène pour observer sa fréquence et ses effets durant le fonctionnement du composant. En sécurité, l'attaquant cible son attaque du mieux qu'il peut. Ceci restreint naturellement la portée de l'analyse aux parties du composant qui assurent des propriétés de sécurité. Cela signifie également que le blindage d'une partie du composant seulement n'augmente pas significativement la sécurité, l'attaquant sélectionnant toujours les parties qui ne sont pas protégées. Par contraste, un phénomène naturel ne « choisit » pas la partie à fauter en fonction du risque posé à la sûreté du système.

2.1.1.4. Nombre de fautes

En sûreté, on ne traite pas le cas des fautes multiples. On considère qu'une seule faute peut se produire pendant l'exécution du programme. Dans [SCS15], essayer de traiter le cas des fautes multiples est même activement découragé. En effet, les auteurs évaluent à 8.821×10^{-27} la probabilité d'observer deux fautes indépendantes au cours d'une même exécution [2], ce que les auteurs considèrent négligeable [3]. Par ailleurs, les auteurs arguent que l'espace de fautes à traiter est trop grand pour supporter les fautes multiples, voire même les fautes simples, en calculant un nombre naïf d'expériences nécessaires de 8.4×10^{15} en faute simple (soit 266 millions d'années-CPU) [4]. Or, comme discuté brièvement à la section 1.2 de ce manuscrit, les attaques par perturbations multiples sont praticables et peuvent constituer des menaces pour la sécurité [KQ07 ; TK10].

1. On peut d'ailleurs lire sous cet angle la quantité importante de travaux effectués en cryptographie : dans ce domaine spécifique, on est capable de bien modéliser les attaques possibles. Un ensemble de propriétés mathématiques des protocoles et des implémentations facilitent les preuves. L'exercice n'est malgré tout pas sans son lot de surprises, comme l'a démontré l'attaque de l'homme du millieu sur Diffie-Hellman qui était jusqu'alors « prouvé sûr » [DH76]. Les attaques par fautes, qui mettent en péril des implémentations de protocoles prouvés sûrs, sont une autre instance de ce phénomène [JT12].

2. Probabilité calculée pour une exécution durant $1s$ d'un système disposant de 1 MiB (2^{23} bits) de mémoire ayant un taux de 0.061 FIT (*Failures In Time*, quantité mesurant le nombre de fautes attendu pendant 10^9 heures de fonctionnement) par MiB.

3. Il faudrait 3.595×10^{18} années pour espérer observer cet événement. Pour référence, on estime à 1.5×10^{10} années l'âge de l'Univers.

4. Pour pouvoir traiter malgré tout le cas des fautes simples qui les intéressent, les auteurs proposent de réduire l'espace de faute à l'aide de techniques telles que le *fault sampling* (estimer la protection du système à partir d'une échantillon choisi aléatoirement de fautes) et le *def/use pruning* (éliminer des classes de fautes dont on peut démontrer aisément l'innocuité parce qu'elles modifient un emplacement mémoire qui sera réécrit avant d'être lu).

2.1.1.5. Détection des fautes

En sûreté comme en sécurité, les programmes doivent détecter les fautes pour pouvoir y réagir. Il existe des moyens de détection matériels en tolérance aux fautes tout comme dans les attaques par perturbation, mais ce ne sont typiquement pas les mêmes. En tolérance aux fautes, on pratique la redondance matérielle, c'est-à-dire qu'un système est dupliqué. Il existe plusieurs moyens de réaliser cette redondance : le système de secours peut prendre le relai quand le système principal échoue, les deux systèmes peuvent fonctionner en même temps et se répartir la charge, on peut également imaginer plusieurs systèmes prenant des décisions à la majorité via des votes [Avi76]. Dans la carte à puce, cette approche de redondance est plus rare pour des raisons de coûts. Une forme de redondance est néanmoins proposée dans [Moo+02 ; Moo+03], où Moore *et al.* promeuvent l'idée d'utiliser la *dual-rail logic*, une méthode d'encodage des valeurs qui doit permettre de masquer les fuites et donc empêcher les attaques par injection de fautes comme les attaques BellCore (et les attaques par canaux cachés). Dans [WW05], Waddle et Wagner décrivent néanmoins des attaques simples pour contrer le *dual-rail logic*, et montrent ainsi que la contre-mesure n'augmente pas le niveau de sécurité de façon significative contre l'injection de fautes.

2.1.1.6. Traitement des fautes

En sûreté, l'objectif est de tolérer la faute, c'est-à-dire d'assurer le fonctionnement du système même en présence d'une faute. En sécurité, la détection d'une faute signifie qu'un attaquant est peut-être en train d'essayer de corrompre le système. Dans cette hypothèse, la réaction appropriée est alors de l'empêcher d'attaquer davantage, ce qui se traduit dans les cartes à puce par des fonctions de déroutement de la puce (la carte cesse l'exécution du programme et devient muette), voire d'autodestruction de la carte.

2.1.2. Évolution des outils d'évaluation en sûreté

De par son histoire plus longue que celle des attaques par perturbation, il est logique qu'on trouve un grand nombre d'outils d'évaluation dédiés à la tolérance aux fautes. On peut classer les outils en trois catégories, suivant le niveau de simulation des fautes : directement sur le système testé, dans un modèle du système écrit en VHDL ou Verilog, ou dans le code du système.

2.1.2.1. Simulation de fautes dans le système

Les outils de cette catégorie sont une évolution directe des toutes premières méthodes de tests de tolérance aux fautes, où une faute était provoquée par l'utilisation d'équipement matériels permettant d'accélérer leur apparition. Ces outils sont directement intégrés dans les systèmes testés et utilisent des mécanismes logiciels (tels que des points d'arrêts ou des interruptions logicielles) pour prendre temporairement la main sur l'exécution, simuler les effets d'une faute en changeant l'état du système, puis rendre la main au code du système. De nombreux outils de cette catégorie sont apparus au tournant des années 90, avec entre autres l'outil FIAT [Seg+88] en 1988, puis FERRARI [KKA92] en 1992 et Xception [CMS98] en 1998. Par rapport à l'injection directe

par l'utilisation d'équipement, ce type d'outils embarqués a l'avantage d'être plus répétable. En revanche, c'est aussi l'introduction d'un inconvénient commun à tous les outils : la nécessité d'utiliser un modèle de faute, qui se doit d'approximer correctement les fautes injectées dans le système.

2.1.2.2. Simulation de fautes dans un modèle du système

En 1994, Jenn *et al.* proposent MEFISTO, un outil d'injection de fautes dans des modèles VHDL [Jen+94]. Ce principe est raffiné en 1997 avec l'outil VERIFY [STB97]. Il s'agit d'injecter des fautes dans une description matérielle bas niveau du circuit, puis d'observer leurs conséquences sur le comportement du circuit. Par rapport à la simulation de fautes dans le système, cette approche a l'avantage de ne pas nécessiter le système (seul un modèle suffit), ce qui est très utile pour pouvoir faire des tests dès la phase de conception du circuit. La nécessité d'établir un modèle complexifie l'utilisation, particulièrement quand le système modélisé a une forte composante logicielle.

2.1.2.3. Simulation de fautes dans le code

Ces outils émulent du code de logiciel embarqué au niveau instruction, ce qui permet de raccourcir la simulation par rapport aux simulations dans un modèle VHDL, mais diminue la précision de la simulation. La simulation des fautes repose sur des techniques de mutation héritées de celles proposée par Lipton *et al.* pour le test logiciel [LDS78]. Ce type de simulation offre également une plus grande indépendance par rapport au système simulé : il n'est pas nécessaire de connaître tous les détails matériels du circuit pour pouvoir effectuer la simulation, la spécification du jeu d'instruction suffit. Ceci permet à des développeurs de tester les systèmes indépendamment des fondeurs. Enfin, ce type d'outils a permis l'introduction de nouveaux modèles d'exécution, comme le modèle d'exécution symbolique, s'inspirant des méthodes formelles telles que la vérification de modèles (*model checking*). Ainsi, dans [LH07], Larsson et Hähnle en 2007 étendent la sémantique d'exécution symbolique à l'injection de fautes qui perturbent un ou plusieurs bits d'un emplacement mémoire. SymPLFIED est, quant à lui, un framework d'injection de fautes, proposé par Pattabiraman *et al.* en 2008 qui utilise la *vérification de modèles* [Pat+08]. Les contributions principales de l'outil sont l'utilisation d'un modèle formel pour représenter des programmes écrits dans un langage d'assembleur générique pour raisonner sur les fautes matérielles et la représentation de tout type d'erreur avec un seul symbole *err*, ce qui permet de diminuer le nombre de chemins à exécuter. On peut également noter que l'outil est parallélisable.

2.1.2.4. Outils indépendants du type de simulation

En 2010, GOOFI est le premier outil d'injection de fautes modulaire, dans lequel l'injection de fautes peut s'effectuer indépendamment de l'infrastructure d'exécution ou de simulation [SBK10]. En 2015, FAIL* reprend cette initiative et vise à proposer un framework complet d'exécution sous injection de fautes, avec un fort focus sur le traitement des résultats. Diverses métriques sont disponibles, avec notamment la possibilité de faire une revue du code en colorant les lignes du code intolérantes aux

fautes [Sch+15]. Cette dernière initiative, *open-source* et adaptable à de nombreuses situations, démontre la maturité des outils d'analyse pour la tolérance aux fautes.

2.2. Outils d'analyse de vulnérabilités à l'injection de fautes

Les défis inhérents au processus d'évaluation de la robustesse à l'injection ont également donné lieu au développement d'outils d'analyse. Leur objectif général est de simuler les effets d'une injection de fautes telle qu'elle peut se produire lors d'une attaque par perturbation afin de pouvoir raisonner sur les composants fautés et diagnostiquer d'éventuelles vulnérabilités. Pour la suite, on précise ce qu'on entend par *vulnérabilité*.

Définition 2.1 *Vulnérabilité*
Faiblesse dans le composant, qui permet à une attaque de réussir.

Par rapport aux outils de tolérance aux fautes, les outils d'analyse de vulnérabilités doivent composer avec un attaquant qui a des objectifs précis. Dans cette section, on décrit les outils existants à notre connaissance, puis on les compare suivant plusieurs critères qu'on a jugés pertinents.

2.2.1. Liste des outils

2.2.1.1. Injection de fautes pour la simulation d'attaque en SystemC

Rothbart *et al.* s'inspirent de MEFISTO [Jen+94] pour proposer dès 2004 un outil d'injection de fautes au niveau des modèles de SystemC [Rot+04]. À notre connaissance, il s'agit du premier outil conçu pour analyser la robustesse face aux attaques par perturbation. SystemC est une bibliothèque `C++` [5] qui permet de modéliser des systèmes au niveau comportemental. On le compare souvent aux langages Verilog et VHDL, bien que SystemC soit moins spécialisé dans la description de composants matériels que ses concurrents. L'outil fonctionne en ajoutant des blocs spéciaux d'injection de fautes entre les différents modules du modèle fonctionnel et les mémoires. Ces blocs interceptent les requêtes d'accès mémoire et renvoient des résultats fautés. Les résultats sont ensuite analysés en comparant les sorties des exécutions fautées avec les sorties nominales.

2.2.1.2. PAFI, simulation de fautes au niveau Verilog

Faurax dans sa thèse de 2008 propose PAFI [Fau08], un outil qui réalise de la simulation de fautes sur des circuits Verilog. PAFI permet de tester les fautes au niveau des portes logiques qui constituent une implémentation matérielle d'algorithme cryptographique. De façon intéressante, Faurax pointe déjà en 2008 les limites de la plupart des simulateurs dynamiques : explosion combinatoire (notamment en fautes multiples indépendantes) et configuration de l'outil complexe à mettre en œuvre.

5. On parle parfois de *langage SystemC*, mais c'est un abus... de langage.

2.2.1.3. `OSCAR`, simulation de code binaire avec modèle de la consommation

Andouard propose en 2009 `OSCAR`, un outil de simulation de code binaire en OCAML conçu spécifiquement pour analyser les attaques par observation [And09, chapitre 4]. L'outil interprète dynamiquement des binaires natif AVR Atmel, et génère un modèle abstrait de la consommation de courant correspondant au code exécuté. Une attaque peut être spécifiée dans un fichier de configuration. On peut ensuite comparer manuellement la consommation sous attaque avec la consommation nominale, et tenter de réaliser une attaque par observation (voir annexe A pour des informations sur ces attaques).

2.2.1.4. SmartCM, mutation de bytecode Java

Machemie *et al.* proposent en 2011 un outil qui simule l'injection de fautes dans des codes Java Card en mutant le bytecode Java [Mac+11]. Chaque mutant est identique au programme d'origine, à la différence qu'un des octets du bytecode a été remplacé par une valeur différente. L'outil génère exhaustivement les mutants de ce type. Il est ensuite capable d'éliminer automatiquement ceux qui sont mal formés (ne correspondent à aucun programme Java légal). L'outil essaie enfin de décompiler automatiquement les mutants générés vers du java, afin qu'un évaluateur puisse observer la sémantique des mutants et vérifier qu'ils ne correspondent pas à une vulnérabilité.

2.2.1.5. Modélisation de l'effet des fautes à haut niveau

Berthomé *et al.* proposent en 2012 un modèle à haut niveau de l'effet des attaques par injection de fautes sur le flot de contrôle [Ber+12], une approche généralisée aux attaques sur le flot de données par Kauffmann-Tourkestansky dans sa thèse de 2012 [Kau12]. La motivation est de permettre à un développeur qui écrit le code à haut niveau (typiquement, en langage `C`) de tester la présence de vulnérabilités dues à l'injection de fautes à bas niveau. L'approche retenue est la mutation de programmes `C` suivant un double modèle d'assignation des variables et de saut exploitant l'instruction `C goto`.

2.2.1.6. TL-FACE, un plugin *Frama-C* pour la preuve d'implémentations cryptographiques

Christofi propose dans sa thèse de 2013 un plugin utilisant *Frama-C* nommé TL-FACE, dans le but de valider des implémentations cryptographiques contre des injections de fautes simples via des méthodes formelles (preuve de théorèmes) [Chr13]. L'analyse s'effectue à partir du code source en `C`. L'utilisateur doit choisir le type d'implémentation (algorithme cryptographique), le modèle de faute et la fonction à vérifier, puis TL-FACE génère ensuite des annotations correspondant à des fautes injectées dans le code source, respectivement à un modèle de faute.

2.2.1.7. Lazart, coloration de graphes de flot de contrôle

Potet *et al.* détaillent en 2014 une approche de vérification d'implémentations par rapport à un modèle de faute basé sur l'inversion de tests conditionnels [Pot+14]. L'approche repose sur un algorithme de coloration du graphe de flot de contrôle du

programme à évaluer, après saisie par l'utilisateur d'un bloc objectif à atteindre ou à éviter, qui permet à l'outil Lazart de générer une stratégie de mutation. À partir de cette stratégie, qui prend en compte les fautes multiples, un mutant est généré puis testé via le moteur d'exécution concolique KLEE [CDE08], ce qui permet la couverture des chemins d'attaque.

2.2.1.8. L'outil `finja`, preuve formelle d'implémentations crypto avec contre-mesure

Rauzy et Guilley proposent en 2014 un outil, `finja`, capable de prouver la robustesse d'implémentations RSA contre l'injection de fautes (en particulier l'attaque BellCore présentée en section 1.2 de ce manuscrit) [RG14a]. L'outil supporte les fautes multiples, même si le nombre de tests effectués par l'outil croit exponentiellement avec le nombre de fautes à injecter. L'outil utilise des conditions de victoire exprimées dans un formalisme qui permet de donner des conditions sur les variables manipulées.

2.2.1.9. Génération d'attaques par faute sur implémentations cryptographiques

Barthe *et al.* proposent en 2014 une approche de génération d'attaques par faute sur des implémentations cryptographiques [Bar+14]. L'objectif est de générer des attaques qui vérifient une propriété mathématique propre à l'implémentation cryptographique visée et qui garantit que les données à protéger (comme les clés privées de l'algorithme, par exemple) soient faciles à récupérer. Par exemple, une telle condition de victoire pour l'algorithme RSA peut être d'obtenir un nombre multiple de p ou q. En effet un tel multiple permet de factoriser simplement le RSA. Cette approche est outillée par une extension d'`EasyCrypt`, un framework de vérification de la sécurité des constructions cryptographiques [Bar+11] (primitives, protocoles). À partir d'une implémentation (non fautée), d'un ensemble de conditions de victoire, un ensemble de fautes, l'outil génère un ensemble d'implémentations fautées qui garantissent les conditions de victoire. L'outil repose sur du calcul de plus faible pré-condition pour la génération des implémentations fautées, et utilise un SMT-solver pour la résolution des contraintes.

2.2.1.10. EFS, Simulateur d'injection de fautes embarqué

Berthier *et al.* proposent en 2014 EFS, un simulateur d'injection de fautes embarqué sur carte [Ber+14]. EFS utilise les mécanismes d'interruptions programmables pour simuler l'injection de fautes suivant un modèle de faute de sauts d'instructions. L'avantage de cette approche est que la simulation d'injection de fautes a lieu directement sur carte, ce qui assure l'exactitude de la sémantique du code exécuté, qui n'est pas garantie dans les approches de simulation de binaire sur ordinateur classique ou FPGA. Cela permet également d'avoir accès aux traces de consommation de courant correspondantes à chaque attaque effectuée, et de travailler ainsi de concert avec les attaques par canaux cachés. En revanche, l'outil doit être embarqué sur chaque modèle de carte à puce, ce qui représente un coût de développement (pour intégrer les mécanismes de simulation dans le code de la carte) et nécessite de pouvoir charger du code sur la carte (« *open-sample* »).

2.2.1.11. Outil d'injection de fautes basé sur QEMU

Holler *et al.* proposent en 2015 un outil d'analyse pour la recherche de vulnérabilités contre les attaques par fautes [Höl+15]. L'outil est basé sur QEMU [Bel05], et est donc compatible avec tous les jeux d'instructions simulés par cette plate-forme. L'objectif est de permettre aux développeurs qui utilisent des COTS de tester des attaques par faute pour vérifier l'absence de vulnérabilités avant de commercialiser leurs produits. L'outil prend en entrée le binaire natif du programme à simuler et un fichier XML qui contient la description des fautes à injecter. Une fois l'attaque effectuée, l'utilisateur analyse manuellement la sortie pour déterminer s'il s'agit d'une vulnérabilité.

2.2.1.12. Outil de simulation via SMT

Goubet *et al.* proposent en 2015 un outil de vérification formelle de contre-mesures contre l'injection de fautes [Gou15]. L'outil prend en entrée deux programmes Thumb-2 [6] fonctionnellement identiques mais de degrés de protection variés contre l'injection de fautes. À partir de cette entrée, il construit un ensemble de prédicats qui peuvent être résolus via un SMT-solver pour vérifier des conditions de victoire. Un traitement manuel des résultats permet ensuite de retrouver des conditions sur l'état initial menant à des vulnérabilités.

2.2.2. Comparaison par approche de détection

Dans cette section, on compare les outils qu'on vient de lister suivant différents critères qui constituent ce qu'on qualifiera leur « approche de détection ». Ces différents critères regroupent les informations de base qui façonnent le périmètre de ce que l'outil est capable de traiter et permettent une classification à haut niveau des différents outils. Parmi ces critères, on a retenu le domaine applicatif (quels types d'exemples l'outil peut-il traiter ?), le point de vue de l'analyse (l'outil se destine-t-il davantage à un fondeur, à un développeur ou à un évaluateur ?), le niveau de l'analyse (sous quelle forme le code est-il traité par l'outil ?), et le niveau d'exécution (comment l'outil exécute-t-il les codes ?).

2.2.2.1. Domaine applicatif

De nombreux outils se destinent explicitement au domaine précis de la cryptographie. C'est le cas des outils [Fau08 ; And09 ; Chr13 ; RG14a ; Bar+14], soit près d'un outil sur deux. L'outil SmartCM [Mac+11] s'applique au cas particulier de Java Card, même s'il ne vise pas de domaine d'application spécifique sur cette plate-forme. Enfin, les autres outils visent un domaine d'application général de sécurité [Rot+04 ; Ber+12 ; Pot+14 ; Ber+14 ; Höl+15 ; Gou15].

2.2.2.2. Point de vue de l'analyse

La figure 1.5 page 14, qui applique le schéma de certification au cas de la carte à puce, illustre l'existence de plusieurs acteurs : le fondeur qui fabrique l'objet carte, le dévelop-

6. Jeu d'instruction ARM condensé sur 16 bits pour la plupart des instructions.

peur qui ajoute son logiciel embarqué, et les CESTI en charge des évaluations. Chacun de ces acteurs a intérêt à disposer d'un outil de détection de vulnérabilités, mais tous n'ont pas exactement les mêmes objectifs, ni les mêmes ressources à disposition. Les fondeurs comme les développeurs ont pour objectif de prévenir l'introduction de vulnérabilités au cours du développement du composant sécurisé, et également de détecter les vulnérabilités existantes le plus tôt possible [7]. Ces objectifs ont pour conséquence que les outils les plus utiles sont ceux qui sont à la disposition des développeurs au cours du développement. Ceci favorise les outils qui s'intègrent à la chaîne de développement, soit directement dans le code source (annotations comme dans TL-FACE [Chr13], macros comme dans [Ber+12]), soit avant ou après la compilation [Höl+15]. Pour les fondeurs, les outils de tests système [Rot+04 ; Fau08] permettent de tester la robustesse à l'injection de fautes avant la réalisation concrète du composant. Par ailleurs, les développeurs disposent nécessairement de composants sur lesquels ils peuvent charger du code (puisqu'ils développent pour ces composants), et ont de plus une bonne maîtrise de ces plate-formes, ce qui leur permet donc d'exploiter cette ressource pour déployer des outils sur carte [Ber+14]. Les objectifs des développeurs favorisent également l'adoption de « bonnes pratiques » de développement et d'outils pour les vérifier automatiquement, qui permettent d'éliminer certaines classes de vulnérabilités sans entrer dans le détail des attaques réalisables ou non, tandis que les CESTI s'intéressent précisément à l'exploitabilité de chaque vulnérabilité découverte, puisque la cotation des attaques rentre dans leurs prérogatives. Enfin, dans le monde académique, les analyses se tournent vers les méthodes formelles et les preuves d'implémentations sous injection de fautes [Chr13 ; Pot+14 ; Bar+14 ; Gou15], particulièrement dans le domaine cryptographique, où l'approche par preuve a rapidement été privilégiée, et où les propriétés mathématiques connues des protocoles permettent de guider l'analyse [Chr13 ; Bar+14].

2.2.2.3. Niveau de l'analyse

L'analyse de vulnérabilités peut s'effectuer à différents niveaux d'abstraction du code. Dans le cadre de la thèse, on a pu en relever quatre.

Le niveau modèles/spécifications. Le niveau modèles/spécifications est le plus haut niveau. C'est un niveau avant l'écriture d'une implémentation proprement dite. Elle permet de modéliser le fonctionnement souhaité du système. L'avantage des modèles est de permettre l'analyse avant même l'écriture d'une implémentation. L'inconvénient est que ce niveau d'analyse ne capte pas toutes les vulnérabilités qui peuvent être introduites par les spécificités de l'implémentation. Dans un contexte d'injection de fautes, ce niveau d'analyse est généralement considéré comme trop abstrait pour donner un diagnostic fiable sur la sécurité du programme considéré. Les outils [Rot+04 ; Bar+14] sont à ce niveau d'analyse.

Le niveau source. Le niveau source est le niveau où les développeurs écrivent l'essentiel du code constituant le logiciel embarqué. En général, dans le domaine des composants sécurisés, le langage utilisé est le `C`. Baser les outils au niveau `C` est donc utile car il

7. Idéalement avant que les cartes destinées à l'évaluation ne soient produites.

s'agit d'un niveau de dialogue direct avec les développeurs. L'outil peut ainsi s'ajouter aisément au flot de développement. Par ailleurs, il existe un bon écosystème d'analyse de code en `C`. Dans le cadre de l'analyse de vulnérabilités, le code source des logiciels à analyser ne sont pas toujours disponibles (par exemple, dans le cas de l'analyse de *malwares*), ce qui peut parfois limiter les cas d'utilisation des outils qui reposent sur ce niveau. Dans la pratique de la certification, le développeur rend son code source disponible à l'évaluateur, et ce problème n'existe pas. Ce niveau permet de détecter les attaques à haut niveau, mais ne dit rien de leur « implémentation » effective en termes de fautes. Parmi les outils présentés, [Ber+12 ; Chr13 ; Pot+14] procèdent à l'analyse au niveau source.

Le niveau assembleur. Le niveau assembleur donne une vision des instructions telles qu'elles sont exécutées par le composant sécurisé. Par ailleurs, l'influence du compilateur a déjà eu lieu : le niveau assembleur est typiquement utilisé par les évaluateurs pour vérifier l'implémentation de certaines contre-mesures. En effet, comme discuté dans [BR10], la phase de compilation crée des différences entre le code source et le code généré, qui peuvent résulter en l'introduction de vulnérabilités dans le code généré, absentes du code source (et qui ne peuvent donc pas être détectées à ce niveau d'analyse). Dans le cadre de l'injection de fautes, ce problème est exacerbé par les contre-mesures, qui sont généralement du code mort dans un modèle d'exécution sans injection de fautes, et donc couramment optimisées par les compilateurs. De plus, certaines parties du code sont directement écrites en assembleur, soit car leur écriture est trop sensible pour être compilée, soit qu'elles utilisent des fonctionnalités exclusivement disponibles dans ce langage (registres ou instructions spécifiques d'un coprocesseur, par exemple). L'inconvénient principal du niveau assembleur est la multiplication du nombre d'instructions par rapport au source, chaque instruction du langage source correspondant à plusieurs instructions assembleur, ce qui allonge de façon conséquente les analyses, en multipliant le nombre de fautes à tester et le nombre d'instructions à interpréter. Ce grain plus fin d'analyse signifie également un plus grand nombre d'attaques détectées, ce qui rend le traitement et l'interprétation des résultats plus complexes. Par ailleurs, le niveau assembleur n'est plus un niveau générique, le langage étant spécifique de chaque jeu d'instruction. Certains outils limitent ce problème en basant leur analyse sur des programmes écrits dans un langage assembleur générique [RG14a]. Cette approche ne fait cependant pas disparaître le problème de portabilité, car il faut alors traduire les langages assembleurs spécifiques vers ce langage assembleur générique (qui devient alors un langage intermédiaire), ou alors écrire directement les programmes dans le langage assembleur générique (ce qui, dans l'état actuel des langages génériques, rend la mise en œuvre impossible dans un composant sécurisé). Outre [RG14a], l'outil présenté dans [And09] est au niveau assembleur Atmel AVR, et [Gou15] est au niveau assembleur Thumb-2.

Le niveau binaire. Le niveau binaire est l'encodage des instructions assembleur en une suite d'octets qui peut être déchiffrée par le processeur du composant sécurisé. Certains outils font le choix de conduire leur analyse directement au niveau binaire, car certains modèles de faute sont dépendants de cet encodage. C'est le cas de [Mac+11]

(pour le bytecode java) et de [Höl+15] (pour les binaires supportés par QEMU). L'outil EFS [Ber+14] injecte des fautes simulées directement dans le code exécuté nativement sur carte, on peut donc parler d'analyse au niveau binaire.

2.2.2.4. Niveau d'exécution du code

Le code analysé par l'outil doit être exécuté suivant un niveau d'exécution, qui varie depuis l'exécution concrète jusqu'à des exécutions symboliques. Parmi les outils listés précédemment, on relève divers niveaux d'exécution.

Embarqué sur carte. EFS [Ber+14] propose un simulateur de fautes embarqué avec le logiciel à tester sur la carte. Dans ce niveau d'exécution, le binaire est exécuté nativement par la carte, le code étant « instrumenté » à bas niveau par le simulateur. L'avantage par rapport à une simulation classique est la précision de l'exécution (puisqu'il n'y a pas de simulation). Il faut en revanche pouvoir charger du code sur la carte à tester, ce qui n'est pas toujours possible pour les évaluateurs.

Simulation dynamique. Lors de la simulation dynamique, les instructions qui composent le code sont interprétées par une machine virtuelle qui maintient un état de la simulation. La simulation dynamique permet l'exécution de modèles haut niveau des circuits [Rot+04 ; Fau08], de codes binaires écrits pour une machine virtuelle [Mac+11] ou pour une autre machine [And09 ; Höl+15]. La simulation dynamique a l'avantage d'un niveau d'abstraction assez faible, ce qui permet des injections de fautes à bas niveau. En revanche, la précision de la simulation n'est pas toujours optimale, car l'émulation logicielle exacte du comportement d'un processeur demande beaucoup d'efforts et de temps de calcul.

Exécution symbolique. L'exécution symbolique d'un programme [Kin76] représente les valeurs des variables par des formules logiques tout au long d'un chemin de code, puis utilise un solveur de contraintes pour trouver des entrées satisfaisant ce prédicat de chemin ou prouvant son infaisabilité. On peut ensuite inverser des parties du prédicat de chemin pour générer de nouvelles entrées couvrant un chemin différent. L'avantage de ce niveau d'exécution est de pouvoir modéliser le comportement de code écrit dans un langage de haut niveau (le code source en `C`, par exemple) [Chr13] ou sur des langages dédiés [RG14a ; Bar+14 ; Gou15], et de pouvoir capter en une seule exécution des informations sur plusieurs valeurs des variables. Un inconvénient est l'explosion combinatoire du nombre de chemins, qui rend difficile le passage à l'échelle de gros programmes à ce niveau d'exécution. Par ailleurs, certains prédicats sont difficiles à résoudre, soit parce qu'ils dépendent de formules elles mêmes difficiles pour les SMT-solver (par exemple, des formules cryptographiques), soit parce qu'ils font intervenir des éléments de l'environnement (fichiers, ressources réseau, fonctions dont le code n'est pas disponible).

Exécution concrète-symbolique. L'exécution *conc*rète-symb*olique* (concolique) [WMM04] est une exécution concrète du code qui effectue une exécution symbolique du chemin

concret parcouru au même moment. Comme dans l'exécution symbolique, cette exécution génère des contraintes, où l'un des prédicats peut être inversé pour générer de nouvelles entrées qui couvrent un autre chemin de code. L'intérêt de l'exécution concrète-symbolique est de lever certaines des limitations de l'exécution symbolique tout en profitant de ses bénéfices. Ainsi, les éléments complexes à résoudre dans les prédicats peuvent être remplacés par leur valeur concrète et ainsi permettre la résolution. L'inconvénient de l'exécution concrète-symbolique est la complexité supplémentaire introduite par le besoin de décider de stratégies pour rendre concrètes ou symboliques des parties de l'exécution. Par ailleurs, il est aussi plus difficile d'établir la garantie que tous les chemins sont effectivement parcourus. Dans l'injection de fautes, l'exécution concrète-symbolique est utilisée par Lazart [Pot+14] via la plate-forme KLEE [CDE08] pour exécuter des chemins fautés et vérifier s'ils permettent d'atteindre l'objectif.

Référence	Nom	Domaine	Niveau	Niveau d'exécution
[Rot+04]	*N/A*	Général	Modèle	Dynamique
[Fau08]	PAFI	Crypto (AES)	Modèle	Dynamique
[And09]	`OSCAR`	Crypto (DPA)	ASM (Atmel)	Dynamique
[Mac+11]	SmartCM	Java Card	Bytecode Java	Dynamique
[Ber+12]	*N/A*	Général	`C`	Dynamique
[Chr13]	TL-FACE	Crypto	`C`	Symbolique
[Pot+14]	Lazart	Général	`C`	Concret-Symbolique
[RG14a]	`finja`	Crypto	ASM générique	Symbolique
[Bar+14]	*N/A*	Crypto	Modèle	Symbolique
[Ber+14]	EFS	Général	Binaire	Embarqué
[Höl+15]	*N/A*	Général	Binaire	Dynamique
[Gou15]	*N/A*	Général	ASM (Thumb-2)	Symbolique

Table 2.1. – Comparaison par approche de détection

2.2.3. Comparaison par modèle de faute

Comme présenté à la section 1.3.2.2 page 23, on a recours à des modèles de faute pour effectuer l'analyse de vulnérabilités. Dans la littérature, on n'observe pas véritablement de modèles de faute unifiés, et le modèle employé varie suivant chaque outil. Par ailleurs, la forme que peuvent prendre les modèles de faute est aussi liée au niveau de l'analyse, les fautes à haut niveau décrivant davantage les effets fonctionnels des fautes, tandis que les fautes à bas niveau décrivent les causes de ces effets. On peut néanmoins classer les modèles de faute en trois catégories, qui se recoupent, suivant qu'ils ont un effet direct sur le flot de données, sur le flot de contrôle, ou sur le code. On n'a pas inclus PAFI [Fau08] dans la comparaison, le modèle de faute employé (glitch sur des portes logiques) étant très éloigné de ceux des autres outils.

2.2.3.1. Modèles de perturbation du flot de données

Les modèles de perturbation du flot de données modifient les données manipulées par le programme. Une distinction classique parmi ces modèles est la durée de la faute [Avi76]. On parle de fautes *volatiles*[8] pour des fautes qui modifient les données pendant un intervalle de temps fixé, avant qu'elles ne reprennent leur valeur d'origine, tandis que les fautes *permanentes* modifient les données jusqu'à ce qu'elles soient réécrites (ou de façon irréversible, si cette opération n'est pas possible). Au niveau source ou supérieur, on parle du modèle d'assignation de variable (*Assign*) [Ber+12 ; Chr13 ; RG14a ; Bar+14 ; Ber+14], qui consiste à assigner une valeur arbitraire à une variable manipulée par le programme en un point du programme (avec, éventuellement, la réassignation ultérieure de sa valeur précédente s'il s'agit d'un modèle de faute volatile). Au niveau assembleur ou inférieur, on parle directement de corruption de registres (*RegCorrupt*) [And09 ; Gou15] ou de mémoire (*DataCorrupt*) [Höl+15].

2.2.3.2. Modèles de perturbation du flot de contrôle

Les modèles de perturbation du flot de contrôle modifient l'ordre dans lequel les instructions du programme sont exécutées. Le plus connu de ces modèles est le modèle du saut d'instruction (*InstSkip*), qui se décline tant au niveau du code source qu'au niveau assembleur. Berthomé *et al.* ont ainsi proposé un modèle basé sur des labels et l'instruction `goto` pour simuler le saut d'instruction au niveau source [Ber+12]. Au niveau assembleur, le modèle du saut d'instruction peut être vu comme le remplacement d'une ou plusieurs instructions par l'instruction `NOP`, qui est l'instruction sans effet[9]. Ce modèle a été étudié par Moro dans sa thèse de 2014 [Mor14], qui a vérifié que 25% des fautes injectées dans un composant ARM 32 bits pouvaient s'interpréter comme relevant du modèle de saut d'instruction. Les outils qui supportent le modèle de saut d'instruction sont [And09 ; Ber+12 ; Bar+14 ; Ber+14 ; Höl+15 ; Gou15]. L'outil Lazart [Pot+14] propose un modèle de faute particulier : l'inversion de test (*BranchChoice*). Au niveau du graphe de flot de contrôle, il s'agit de forcer le passage d'une condition à vraie quand elle devrait être fausse (ou à fausse au lieu de vraie).

2.2.3.3. Modèles de remplacement d'instruction

Les modèles de remplacement d'instruction (*InstRepl*) s'appliquent au niveau assembleur ou inférieur. Il s'agit de remplacer une ou plusieurs des instructions assembleur qui constituent le programme par d'autres. La connaissance de l'encodage binaire est nécessaire pour faire le lien avec des modèles de corruption mémoire, et remplacer les instructions en fonction des octets modifiés en mémoire. Ce modèle de faute a des conséquences qui peuvent être difficiles à décrire sur l'exécution du code, avec des sauts non prévus par le graphe de flot de contrôle, et l'exécution décalée d'instructions. Ce modèle de faute est implémenté par les outils [Rot+04 ; Mac+11 ; Höl+15].

8. on trouve également le terme « transient » dans la littérature

9. Certains outils [Ber+12 ; Ber+14] autorisent également de décrémenter le registre d'instruction, ce qui peut s'apparenter à des sauts en arrière.

Référence	Nom	InstSkip	InstRepl	BranchChoice	Assign	RegCorrupt	DataCorrupt
[Rot+04]	*N/A*	✗	✓	✗	✗	✗	✓
[Fau08]	PAFI	—	—	—	—	—	—
[And09]	`OSCAR`	✓	✗	✗	✗	✓	✗
[Mac+11]	SmartCM	✗	✓	✗	✗	✗	✗
[Ber+12]	*N/A*	✓	✗	✗	✗	✗	✗
[Chr13]	TL-FACE	✗	✗	✗	✓	✗	✗
[Pot+14]	Lazart	✗	✗	✓	✗	✗	✗
[RG14a]	`finja`	✗	✗	✗	✓	✗	✗
[Bar+14]	*N/A*	✓	✗	✗	✓	✗	✗
[Ber+14]	EFS	✓	✗	✗	✓	✗	✗
[Höl+15]	*N/A*	✓	✓	✗	✗	✗	✓
[Gou15]	*N/A*	✓	✗	✗	✗	✓	✗

TABLE 2.2. – Comparaison par modèle de faute

2.2.4. Comparaison par campagnes d'injection

Dans cette thèse, on définit comme *campagne d'injection* l'ensemble du paramétrage d'un outil déterminant le périmètre de l'évaluation. Dans cette catégorie, on retient trois facteurs pertinents : les entrées utilisateur, l'état initial de l'exécution, et la liste des fautes à injecter.

2.2.4.1. Gestion des entrées utilisateur

En fonction des entrées utilisateur, le résultat de la simulation est différent. Par exemple, une exécution de `VerifyPIN` avec une valeur de PIN fausse doit refuser l'authentification, tandis qu'avec la bonne valeur du PIN, l'authentification doit s'effectuer. Dans l'injection de fautes, on peut imaginer que certaines entrées spécifiques (certaines valeurs de PIN fausses) puissent permettre des attaques impossibles pour les autres valeurs. Cette dimension du problème n'est pas traitée par les outils dynamiques, qui travaillent généralement sur une trace déterminée par une entrée fixée par l'utilisateur [Rot+04 ; And09 ; Mac+11 ; Ber+12 ; Ber+14 ; Höl+15], tandis que les outils qui ont un modèle d'exécution symbolique peuvent traiter les entrées par classe [Chr13 ; Pot+14 ; RG14a ; Bar+14 ; Gou15].

2.2.4.2. État initial de l'exécution

L'état initial de l'exécution désigne l'état du composant sécurisé, en particulier de la mémoire, au début d'une exécution. Comme pour la gestion des entrées utilisateur, cet élément de configuration est important pour les outils car il conditionne en partie les résultats de la campagne d'injection, et il permet la simulation. En effet, si un outil embarqué comme EFS [Ber+14] ne nécessite que peu de configuration (la même que pour lancer la commande sur carte), la simulation dynamique d'un composant nécessite de pouvoir modéliser le contexte d'exécution de la commande. Par ailleurs, Lazart [Pot+14] nécessite une phase d'initialisation pour savoir quelles variables rendre

symboliques et de quelle manière, et TL-FACE [Chr13] utilise un système d'annotations pour décrire des préconditions sur l'état.

2.2.4.3. Liste des fautes à injecter

Les outils génèrent des exécutions fautées suivant une liste des différentes fautes à injecter. La gestion de cette liste s'effectue suivant deux méthodes distinctes parmi les outils. Certains outils demandent explicitement de spécifier les fautes à injecter via un fichier de description ou un scénario. C'est le cas des outils [Rot+04 ; Fau08 ; And09 ; Ber+14 ; Höl+15]. L'autre méthode, suivie par [Mac+11 ; Ber+12 ; Chr13 ; RG14a ; Bar+14 ; Gou15], est de générer de façon exhaustive les fautes possibles, relativement à un modèle de faute. Cette approche a pour avantage de ne pas laisser de côté un cas qui pourrait être problématique, mais pour inconvénient de faire croître énormément le nombre de résultats à traiter par la suite (et le temps d'exécution). Parmi les outils, Lazart [Pot+14] a une méthode particulière : l'utilisateur configure l'outil, mais pour spécifier un objectif. Les fautes à injecter ne doivent donc pas être spécifiées directement par l'utilisateur, mais sont inférées par l'objectif décrit.

2.2.4.4. Gestion des fautes multiples

Certains outils prévoient spécifiquement la possibilité de gérer les fautes multiples indépendantes. Ainsi, Lazart [Pot+14] peut injecter de façon exhaustive un nombre non borné de fautes, une performance rendue possible par le modèle d'exécution symbolique et la réduction des cas d'injection de fautes par l'algorithme de coloration. L'outil `finja` [RG14a] et celui décrit dans [Höl+15] supportent un nombre n de fautes, mais la complexité de l'analyse est exponentielle en le nombre de fautes injectées, ce qui limite fortement l'analyse en pratique. `OSCAR` supporte explicitement les double fautes. Les autres outils ne semblent pas supporter les fautes multiples.

Référence	Nom	Entrées	Liste des fautes	Fautes multiples
[Rot+04]	*N/A*	Fixée	Utilisateur	✗
[Fau08]	PAFI	Fixée	Utilisateur	✗
[And09]	`OSCAR`	Fixée	Utilisateur	2 fautes
[Mac+11]	SmartCM	Fixée	Exhaustive	✗
[Ber+12]	*N/A*	Fixée	Exhaustive	✗
[Chr13]	TL-FACE	Variable	Exhaustive	✗
[Pot+14]	Lazart	Variable	Exhaustive	✓
[RG14a]	`finja`	Variable	Exhaustive	n fautes
[Bar+14]	*N/A*	Variable	Exhaustive	✗
[Ber+14]	EFS	Fixée	Utilisateur	✗
[Höl+15]	*N/A*	Fixée	Utilisateur	n fautes
[Gou15]	*N/A*	Variable	Exhaustive	✗

TABLE 2.3. – Comparaison par campagne d'injection

2.2.5. Comparaison par méthode de traitement des résultats

Le traitement des résultats est l'opération effectuée sur les résultats de l'analyse, et qui vise à répondre à la question initiale de robustesse du code considéré. Suivant l'objectif de l'analyse, différents types de critères peuvent être intéressants : pour les développeurs qui choisissent les contre-mesures adaptées, il est intéressant de développer un critère de robustesse permettant de déterminer la résistance de chaque partie du code (idéalement, des lignes du code source) aux attaques. À l'inverse, les évaluateurs souhaitent émettre un diagnostic global sur le niveau de sécurité du code. De tels critères sont difficiles à mettre en place : l'utilisation de modèles de faute tend à produire beaucoup de codes fautés possibles, qu'il faut donc classer.

Parmi les outils présentés, on relève quatre méthodes de traitement des résultats.

2.2.5.1. Traitement manuel

Dans la méthode de traitement manuel, on utilise l'expertise des évaluateurs pour conclure, c'est-à-dire que les implémentations fautées sont générées, et éventuellement jouées, mais c'est l'examen par un expert qui détermine son niveau de dangerosité. Cette méthode est employée par les outils [Fau08 ; And09 ; Höl+15].

2.2.5.2. Traitement par comportement

Certains outils ([Rot+04 ; Mac+11 ; Ber+12 ; Ber+14]) effectuent un pré-traitement des résultats en les classant en fonction du comportement observé, suivant que :

— l'exécution s'est terminée sans erreur, avec le même résultat que l'exécution non fautée ;
— l'exécution s'est terminée sans erreur, mais avec un résultat différent de l'exécution non fautée [10] ;
— l'exécution ne s'est pas terminée (erreur).

Ceci offre un premier pré-traitement des résultats, également observable sur carte, qui reste cependant imprécis (notamment, ne tient pas compte des effets de bord sur le système).

2.2.5.3. Traitement par métriques d'évaluation

Certains outils génèrent des métriques d'évaluation, c'est-à-dire des quantités qui se veulent représentatives de la robustesse du code.

Dans [Ber+12], les auteurs proposent de classer la proportion d'attaques réussies en fonction de l'adresse attaquée. Cette métrique locale donne une première vision des adresses les plus vulnérables. Dans [Pot+14] et [Ber+14], les auteurs dénombrent le nombre d'attaques réussies après analyse afin de comparer plusieurs implémentations d'une même fonctionnalité en fonction du nombre d'attaques pour chaque implémentation. [Rot+04] propose un critère proche, en calculant le rapport du nombre d'attaques réussies sur le nombre d'attaques effectuées.

10. Les attaques réussies se trouvent donc parmi les exécutions de ce type.

2.2.5.4. Traitement par preuve

Les outils qui font du traitement par preuve visent à démontrer des propriétés particulières du programme évalué. En lien avec la robustesse, il peut s'agir de démontrer qu'une variable contenant des informations sensibles ne peut pas être lue par un attaquant, quelle que soit la faute considérée (dans un modèle de faute). Les outils qui font du traitement par preuve sont [Chr13 ; Pot+14 ; RG14a ; Bar+14 ; Gou15].

2.2.6. Comparaison par méthode de guidage des résultats

La particularité de l'outil [Ber+14] est de s'exécuter directement sur le composant dont on souhaite évaluer le code. Cette particularité permet d'effectuer des acquisitions sur les canaux auxiliaires pendant les campagnes d'injection. Ces acquisitions peuvent ensuite être utilisées comme un guide indirect pour les attaques par perturbation : il est possible en attaque réelle de faire varier les paramètres d'attaque jusqu'à tenter de reproduire les courbes obtenues pendant la campagne d'injection.

Référence	Nom	Manuel	Comportement	Métrique	Preuve
[Rot+04]	*N/A*	✗	✓	✓	✗
[Fau08]	PAFI	✓	✗	✗	✗
[And09]	`OSCAR`	✓	✗	✗	✗
[Mac+11]	SmartCM	✗	✓	✗	✗
[Ber+12]	*N/A*	✗	✓	✓	✗
[Chr13]	TL-FACE	✗	✗	✗	✓
[Pot+14]	Lazart	✗	✗	✓	✓
[RG14a]	`finja`	✗	✗	✗	✓
[Bar+14]	*N/A*	✗	✗	✗	✓
[Ber+14]	EFS	✗	✓	✓	✗
[Höl+15]	*N/A*	✓	✗	✗	✗
[Gou15]	*N/A*	✗	✗	✗	✓

TABLE 2.4. – Comparaison par traitement des résultats

2.3. Conclusion

2.3.1. Récapitulatif des outils d'analyse de vulnérabilités

La table 2.5 récapitule les caractéristiques des différents outils discutées à la section précédente afin de donner une vision globale de l'existant.

Référence (nom)	Approche de détection			Modèle de faute					
	Domaine	Niveau	Exécution	InstSkip	InstRepl	BranchChoice	Assign	RegCorrupt	DataCorrupt
[Rot+04]	Général	Modèle	Dynamique	✗	✓	✗	✗	✗	✓
[Fau08] (PAFI)	Crypto	Modèle	Dynamique	—	—	—	—	—	—
[And09] (`OSCAR`)	Crypto	ASM	Dynamique	✓	✗	✗	✗	✓	✗
[Mac+11] (SmartCM)	Java Card	Bytecode	Dynamique	✗	✓	✗	✗	✗	✗
[Ber+12]	Général	`C`	Dynamique	✓	✗	✗	✗	✗	✗
[Chr13] (TL-FACE)	Crypto	`C`	Symbolique	✗	✗	✗	✓	✗	✗
[Pot+14] (Lazart)	Général	`C`	Concolique	✗	✗	✓	✗	✗	✗
[RG14a] (`finja`)	Crypto	ASM	Symbolique	✗	✗	✗	✓	✗	✗
[Bar+14]	Crypto	Modèle	Symbolique	✓	✗	✗	✓	✗	✗
[Ber+14] (EFS)	Général	Binaire	Embarquée	✓	✗	✗	✓	✗	✗
[Höl+15]	Général	Binaire	Dynamique	✓	✓	✗	✗	✗	✓
[Gou15]	Général	ASM	Symbolique	✓	✗	✗	✗	✓	✗

Référence (nom)	Campagne d'injection			Traitement des résultats			
	Entrées	Liste des fautes	Fautes multiples	Manuel	Comportement	Métrique	Preuve
[Rot+04]	Fixée	Utilisateur	✗	✗	✓	✓	✗
[Fau08] (PAFI)	Fixée	Utilisateur	✗	✓	✗	✗	✗
[And09] (`OSCAR`)	Fixée	Utilisateur	2 fautes	✓	✗	✗	✗
[Mac+11] (SmartCM)	Fixée	Exhaustive	✗	✗	✓	✗	✗
[Ber+12]	Fixée	Exhaustive	✗	✗	✓	✓	✗
[Chr13] (TL-FACE)	Variable	Exhaustive	✗	✗	✗	✗	✓
[Pot+14] (Lazart)	Variable	Exhaustive	✓	✗	✗	✓	✓
[RG14a] (`finja`)	Variable	Exhaustive	n fautes	✗	✗	✗	✓
[Bar+14]	Variable	Exhaustive	✗	✗	✗	✗	✓
[Ber+14] (EFS)	Fixée	Utilisateur	✗	✗	✓	✓	✗
[Höl+15]	Fixée	Utilisateur	n fautes	✓	✗	✗	✗
[Gou15]	Variable	Exhaustive	✗	✗	✗	✗	✓

TABLE 2.5. – Tableau récapitulatif des outils

2.3.2. Une multitude d'outils

On constate donc l'apparition d'une multitude d'outils depuis les années 2000, et en particulier depuis 2010. Ceci démontre l'importance actuelle du sujet de l'analyse automatique des vulnérabilités contre l'injection de fautes, qu'on peut tenter d'expliquer par différents facteurs concordants. L'analyse de vulnérabilités prend également de l'élan au sein de la sécurité informatique traditionnelle, à une époque où le domaine dans son entier est très valorisé suite aux nombreuses affaires de piratage informatique et aux révélations relatives à l'espionnage par la NSA. Les agences nationales telles que l'ANSSI jouent également un rôle incitatif à l'outillage pour l'analyse de vulnérabilités tant chez les évaluateurs que chez les développeurs. Enfin, les méthodes d'injection de fautes évoluent avec des équipements, notamment électromagnétiques, qui sont de plus en plus abordables et de plus en plus précis, multipliant par là le potentiel de nuisance de l'attaquant.

Les outils ne sont, sauf exception [Bar+14 ; Gou15], pas disponibles, ni en *open-source*, ni même commercialement. Il s'agit d'outils industriels perçus comme du savoir-faire et comme des avantages compétitifs, ce qui heurte malencontreusement la reproductibilité des recherches effectuées, d'autant que les exemples de codes sur lesquels ils sont testés ne sont souvent pas disponibles non plus (code propriétaire des développeurs), et les limites de l'injection de fautes ne sont pas clairement identifiées (qu'il s'agisse du modèle de faute le plus pertinent à utiliser, ou du problème des fautes multiples).

2.3.3. Entre tolérance et perturbation

À la lumière des états de l'art respectifs de la tolérance aux fautes et des attaques par perturbation, on peut formuler plusieurs remarques. Tout d'abord, l'histoire de la simulation en tolérance aux fautes commence plus tôt que celle des attaques par perturbation. Ceci a donné un peu « d'avance » au premier domaine par rapport au second.

En dépit de cette « avance », les outils de chaque domaine ont exploré des approches globalement similaires : injection de fautes au niveau système sur des modèles VHDL/-Verilog/SystemC, injection de fautes au niveau jeu d'instruction, exploration de l'exécution symbolique autour de 2010. On peut noter le fait intéressant que les attaques par canaux cachés semblent avoir relancé l'intérêt de simulateurs embarqués sur carte, comme le démontrent les auteurs de [Ber+14]. Une différence importante entre la sûreté et la sécurité est l'existence du sous-domaine de la cryptographie, qui a fortement influencé le développement des outils d'analyse contre les attaques par perturbation. La plupart des approches par preuves se positionnent ainsi dans ce sous-domaine. Il serait intéressant d'emboîter le pas à la tolérance aux fautes et d'étendre l'injection symbolique de fautes proposée dans [LH07] et [Pat+08] aux attaques par perturbation. De même, la plate-forme FAIL* [Sch+15], *open-source*, indépendante de l'infrastructure d'exécution et disposant de véritables analyses des résultats se pose comme un modèle à poursuivre pour les outils d'analyse de vulnérabilités. De telle plate-formes seraient souhaitables pour les attaques par perturbation pour pouvoir proposer des outils véritablement polyvalents.

En revanche, le domaine de la tolérance aux fautes se désintéresse des fautes multiples

indépendantes, qui ont une probabilité d'occurrence naturelle négligeable. En perturbation, ce scénario ne peut cependant pas être ignoré, car comme discuté à la section 1.2 du chapitre 1, l'attaquant a la possibilité de le faire. Ceci pousse à développer de nouvelles analyses afin de maîtriser l'explosion combinatoire qui en résulte. Le résultat de ces recherches pourrait constituer un apport en tolérance aux fautes pour maîtriser la combinatoire : en effet dans ce domaine on considère des fautes partout dans le système, spatialement et temporellement, ce qui donne lieu à des espaces d'injection de fautes typiquement beaucoup plus grands que ceux considérés en attaques par perturbation en faute simple, où l'analyse se restreint aux seules fonctions de sécurité.

Autre grande différence, le traitement des résultats. En effet, en sûreté, le danger posé par une injection de fautes est défini très précisément en termes de crash ou de mauvais fonctionnement du système. Quand on considère les attaques par perturbation, l'attaquant souhaite obtenir un des biens protégés par le composant sécurisé en compromettant une ou plusieurs de ses propriétés de sécurité. La présence d'un attaquant rend beaucoup plus difficile la vérification qu'il n'existe pas de menace. Les outils qui traitent de la cryptographie exploitent les propriétés mathématiques des algorithmes et les attaques connues pour démontrer l'absence de faille par rapport à un modèle de faute donné. La plupart des autres outils n'offrent en revanche guère mieux qu'un pré-traitement des résultats (éliminant les cas identiques aux cas nominal et les cas d'erreur), laissant aux experts le soin d'analyser manuellement l'exploitabilité des fautes injectées.

2.3.4. Un outil pour les évaluateurs ?

Dans les différents points de vue adoptés par les outils d'analyse de vulnérabilités à l'injection de fautes, on note que le point de vue des évaluateurs des CESTI n'est pas couvert [11].

L'élaboration d'un outil destiné à un CESTI suppose plusieurs conditions particulières. L'outil doit être facilement portable d'une plate-forme à l'autre, il ne peut pas nécessiter l'exécution embarquée, les CESTI n'ayant pas toujours la possibilité de charger du code sur les cartes, il doit permettre la répétabilité des expériences et le calcul de métriques de traitement des résultats pour diagnostiquer la robustesse. Au cours de la thèse, j'ai été amené à implémenter CELTIC, un outil d'analyse de vulnérabilités à l'injection de fautes que j'ai spécialement conçu pour le CESTI du CEA-LETI. Le chapitre 4 revient en détail sur les choix de conception qui ont gouverné au développement de cet outil.

Par ailleurs, les différents outils étudiés manipulent des modèles de faute très variés, allant du remplacement d'instructions à l'assignation de variables, à des niveaux d'analyse très différents, des spécifications au binaire, sans en général en questionner la pertinence par rapport au composant ciblé et à l'équipement d'attaque. Dans le chapitre 3, nous proposons une approche qui permet de dériver des modèles de faute liés au composant ciblé et à l'équipement d'attaque, modèles que notre outil accepte en entrée de ses analyses.

11. C'est en tout cas valable pour les outils dont on a connaissance.

Chapitre 3

Inférence de modèles de faute

Je sais que deux et deux font quatre—[...]—mais je dois dire que si par un quelconque moyen je pouvais convertir deux et deux en *cinq*, ce serait pour moi un bien plus grand plaisir.

(Lord George Gordon Byron, Alas! the Love of Women: 1813-1814)

Résumé du chapitre

Ce chapitre présente l'inférence de modèle de faute, une nouvelle méthode pour déterminer quelles fautes sont réalisables en attaquant un certain composant avec un équipement spécifique d'attaque. Il débute par un retour sur le processus d'évaluation de la robustesse à l'injection de faute, constitué de deux sous-processus. On montre que, s'il est souhaitable de pouvoir lier les résultats de ces deux sous-processus, ce n'est pas simple en pratique car les fautes qu'ils manipulent sont caractérisées d'une part par des paramètres de l'équipement d'attaque, d'autre part par des paramètres du code. Pour remédier à ce problème fondamental on propose la notion clé de modèle de faute probabiliste, qui associe des paramètres d'équipement à des paramètres de code et attribue une probabilité d'occurrence à cette association. On détaille ensuite la méthode d'inférence de modèles de faute, qui vise à construire de tels modèles probabiliste à partir d'attaques par perturbation répétées sur carte, mettant en jeu des programmes particuliers appelés programmes de détection de faute. On applique cette méthode à un cas d'étude, la carte C, et on donne les modèles probabilistes correspondant avec un équipement d'attaque électromagnétique et un équipement laser. Enfin, on conclut en donnant plusieurs modèles de fautes inférés grâce à cette méthode et en discutant la pertinence, et les applications.

Sommaire

3.1. Introduction

On a vu à la section 1.3.3 du chapitre 1 que le processus d'évaluation de la robustesse à l'injection de fautes s'articule suivant deux sous-processus : une revue de code, et des tests de pénétration.

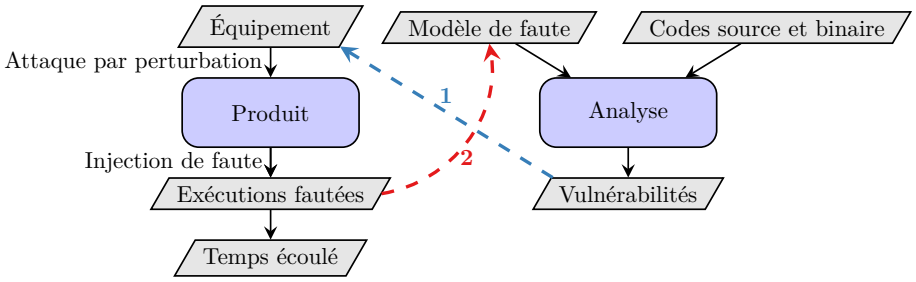


FIGURE 3.1. – Liens manquants

Dans la théorie, on utilise la sortie de la revue de code pour guider les tests de pénétration. Dans la pratique, on constate que ce n'est pas chose facile, les deux sous-processus étant en effet très séparés : difficile d'établir avec certitude qu'une vulnérabilité découverte pendant la revue de code correspond bien à une vulnérabilité observée pendant les tests de pénétration, ou bien s'il s'agit de deux vulnérabilités différentes.

On a identifié deux problématiques autour de ces disparités, numérotées 1 et 2 sur la figure 3.1, qui illustre les liens manquants entre les deux sous-processus :

1. Les vulnérabilités obtenues durant la revue de code s'expriment en fonction d'une faute injectée à une adresse donnée dans un emplacement mémoire donné. Par contraste, les fautes injectées pendant les tests de pénétration s'expriment en fonction du paramétrage de l'équipement d'attaque utilisé. **Comment lier les paramètres du code et ceux de l'équipement d'attaque ?**
2. Durant la revue de code (manuelle, ou via l'analyse de code avec un outil), le modèle de faute utilisé est choisi arbitrairement. Dans la pratique, au CESTI, on considère généralement les modèles de faute de saut d'instruction et d'inversion de test. **Comment s'assurer que le modèle de faute utilisé durant la revue de code capte bien les fautes injectables sur carte, et seulement celles-ci ?**

L'objectif de ce chapitre est d'apporter des réponses à ces deux problématiques, par l'étude des deux espaces de paramètres d'attaque et via une nouvelle méthode d'inférence de modèles de faute.

3.1.1. Deux espaces de paramètres

Dans cette section, on détaille donc les deux espaces de paramètres : l'espace des paramètres d'équipement qui caractérise les tests de pénétration, et l'espace des paramètres de code qui caractérise la revue de code.

3.1.1.1. Espace des paramètres d'équipement

Comme détaillé à la section 1.2, les attaques par perturbation utilisent un équipement spécialisé tel que le générateur de glitches, le laser ou l'injecteur électromagnétique (voir section 1.2.2). Pour effectuer des attaques, l'attaquant doit choisir la valeur de certains paramètres de l'équipement utilisé : par exemple, pour un laser, il faut choisir la position (x, y) du laser dans le plan de la carte ciblée, la largeur s (*spot*) de la

tâche d'éclairement, le délai d (*delay*) après lequel l'éclairement commence, la durée w (*width*) de l'éclairement et l'intensité I de l'éclairement. L'ensemble des valeurs que peuvent prendre ces paramètres constituent l'espace des paramètres d'équipement (ici, pour le laser).

Définition 3.1 *Espace des paramètres d'équipement*
Ensemble noté $\mathcal{P}$ dont les éléments sont des listes de valeurs pour chacun des paramètres de l'équipement.

Par exemple, en reprenant les paramètres définis pour le laser, un élément $p \in \mathcal{P}$ peut s'écrire [1] : $p \hat{=} (x = 12\,\mu\text{m},\ y = 24\,\mu\text{m},\ s = 8\,\mu\text{m}^2,\ d = 3800\,\text{ns},\ w = 850\,\text{ns},\ I = 80\,\text{W})$. Afin d'alléger les notations, le nom et les unités des paramètres peuvent être omis s'ils ont été précisés auparavant. Une faute effectuée durant les tests de pénétration est donc exprimée suivant le valeurs des paramètres d'équipement au moment où la faute a été injectée. On dira que la faute f est de paramètre p dans l'espace $\mathcal{P}$. Par exemple, au laser, on peut avoir une faute de paramètre $p \hat{=} (12, 24, 8, 3800, 850, 80)$. On choisit typiquement de faire figurer les paramètres d'équipement qui sont sous le contrôle de l'attaquant et qu'il peut ainsi faire varier. Dans ce manuscrit on fait le choix d'exclure la longueur d'onde des lasers, la puissance transmise par notre injecteur électromagnétique et la température de la pièce, simplement car nos expériences ne sont pas *paramétrables* suivant ces données [2]. De même, on se permet de réduire la liste des paramètres considérés quand certains d'entre eux voient leur valeur fixée à une constante (par exemple, si on fixe la position (x, y) dans une expérience, on ne rappellera pas systématiquement leur valeur pour chaque élément p de $\mathcal{P}$).

Dans la pratique, on peut partager les différents paramètres d'équipement en plusieurs catégories, susceptibles de se recouper :

- Une dimension **spatiale** qui regroupe les paramètres qui ont un rapport avec la position entre l'équipement et la cible. On y retrouve x, y, mais aussi pour le laser la largeur de spot s. En électromagnétique, on trouve aussi l'angle θ entre la boucle de l'injecteur et la carte, et l'altitude z entre la pointe de l'injecteur et la carte.
- Une dimension **temporelle** qui regroupe les paramètres qui ont un rapport au temps. On y retrouve le délai d et la durée w.
- Une dimension **énergétique** qui regroupe tous les paramètres ayant trait à l'énergie injectée par la perturbation. On y trouve l'intensité lumineuse I pour le laser. On pourrait également y classer s, z et w, qui sont déjà dans les dimensions spatiale ou temporelle, car ils ont une influence sur l'énergie transmise par la perturbation.

L'appartenance d'un paramètre à une dimension ou à une autre est connue pour déterminer en partie l'effet du paramètre sur la faute injectée au cours de l'attaque. Ainsi,

1. Les valeurs numériques sont données à titre indicatif et ne représentent pas des données d'expériences réelles.
2. On pourra toutefois ponctuellement les donner à titre informatif et pour améliorer la reproductibilité des expériences.

un paramètre spatial a une influence sur la nature de la faute injectée : faute mémoire, faute sur les registres, etc. Un paramètre temporel tend à avoir une influence sur le moment où la faute est injectée : tôt ou tard dans le code. Un paramètre énergétique tend à avoir un impact sur la probabilité d'injecter la faute. Ces observations empiriques seront étayées dans la section 3.4.

3.1.1.2. Espace des paramètres de code

Les évaluateurs qui effectuent une revue de code pour l'analyse de robustesse à l'injection de faute font des déclarations du type :

« Si j'injecte la faute qui renvoie 42 lorsque le registre A est lu à la troisième instruction, alors [...] »

« La faute qui assigne 42 à la variable x à la troisième ligne de la première boucle est détectée par cette contre-mesure. »

Comme pour les fautes injectées durant les tests de pénétration, les fautes considérées par les évaluateurs sont caractérisées par plusieurs paramètres. Ces paramètres sont exprimés en fonction du niveau de l'analyse (modèle, code source, assembleur, binaire, voir 2.2.2). Il n'existe pas à notre connaissance de formalisation permettant de décrire les paramètres de code qui caractérisent une faute. On propose une formalisation générique pour décrire ces paramètres. Cette formalisation est utilisée par notre outil de simulation et d'injection de fautes CELTIC, dont la présentation fait l'objet du chapitre 4. Les instructions exécutées par le processeur sont décomposées en un ensemble d'opérations atomiques de lecture et d'écriture d'emplacements mémoire (registre du processeur, case mémoire adressée). Une trace d'exécution est alors une liste de ces opérations, associée à une exécution d'un programme.

Dans ce contexte, une faute est décrite par l'ajout ou la modification d'une des opérations effectuées dans la trace d'exécution. Un paramètre c est donc de l'une des quatre formes suivantes : $(i, \text{store}(\ell, b))$, $(i, \text{read}(\ell))$, $(i, \text{store}_m(\ell, a, b)$ ou $(i, \text{read}_m(\ell, a, b)$. On les détaille :

- $(i, \text{store}(\ell, b))$ ajoute une opération d'écriture de la valeur b à l'emplacement mémoire ℓ entre l'opération numérotée $i-1$ et l'opération numérotée i.
- $(i, \text{read}(\ell))$ ajoute une opération de lecture de l'emplacement ℓ entre l'opération numérotée $i-1$ et l'opération i.
- $(i, \text{store}_m(\ell, a, b))$ modifie l'opération d'écriture numérotée i pour écrire dans ℓ la valeur b à la place de la valeur a. Préciser la valeur a permet de n'effectuer la faute que si a est la valeur normalement écrite.
- $(i, \text{read}_m(\ell, a, b))$ modifie l'opération de lecture numérotée i pour lire la valeur b à la place de la valeur a contenue dans ℓ[3].

Cette formalisation permet de couvrir les fautes qui se produisent durant le fonctionnement du processeur à bas niveau. Elle n'est pas adaptée à une description à plus haut niveau (source).

Comme pour les paramètres d'équipement, on peut caractériser les paramètres de code suivant plusieurs dimensions :

3. La valeur contenue dans ℓ n'est pas modifiée par cette faute.

— La dimension **spatiale** qui regroupe les paramètres désignant l'emplacement mémoire visé par la faute. Dans notre formalisation, c'est le paramètre ℓ. Par exemple, sur du code assembleur, le registre A ou l'adresse mémoire `0x41ff`. Sur du code source, la variable x, la condition c.
— La dimension **temporelle** qui regroupe les paramètres indiquant le moment où la faute se produit. Dans notre formalisation, c'est le paramètre i. Sur du code assembleur, il peut s'agir du numéro d'une instruction dans une trace d'exécution, d'un numéro d'opération dans un simulation par un outil, etc. Sur du code source, on considère de la même façon le numéro de *statement*[4] dans une trace d'exécution.
— La dimension **qualitative** qui donne des informations sur la valeur injectée par la faute. Dans notre formalisation, c'est la nature de l'opération (read, read_m, store, store_m) et ses paramètres (a, b).

De façon similaire à l'espace des paramètres d'équipement, on définit l'espace $\mathcal{C}$ des paramètres du code.

Définition 3.2 *Espace des paramètres du code*
Ensemble noté $\mathcal{C}$ dont les éléments sont des listes des caractéristiques associées à une faute durant la revue de code.

3.1.1.3. Lier $\mathcal{P}$ et $\mathcal{C}$

La problématique de lier les paramètres de l'équipement et ceux du code peut s'exprimer avec $\mathcal{P}$ et $\mathcal{C}$. Pour une faute exprimée suivant $p \in \mathcal{P}$ (respectivement, $c \in \mathcal{C}$), il s'agit de trouver le paramètre $c_p \in \mathcal{C}$ (respectivement, $p_c \in \mathcal{P}$) exprimant la même faute.

Pour les paramètres qui appartiennent à la dimension temporelle dans chacun des deux espaces, les évaluateurs peuvent approximativement établir ce lien en utilisant des points de synchronisation, c'est-à-dire des points temporels qu'il est possible de reconnaître à la fois dans la trace d'exécution dans les signaux comme la consommation. Par exemple, les écritures en mémoire non volatile sont identifiables dans le code (il suffit de connaître quelles variables sont non volatiles au niveau source, ou pour l'assembleur de savoir que l'adresse écrite se situe dans une zone de mémoire non volatile), et correspondent souvent à des motifs très reconnaissables sur les traces de consommation. En observant entre quelles écritures une faute se manifeste dans la revue de code et dans les tests de pénétration, on peut ainsi donner un intervalle de valeurs pour les parties temporelles de c_p ou p_c. En revanche, ce type de lien est plus complexe à établir pour les autres dimensions, en particulier pour la dimension qualitative de $\mathcal{C}$.

3.1.2. Approche proposée

Pour répondre aux deux problématiques introduites dans ce chapitre (voir figure 3.1), on propose une approche d'inférence de modèles de faute, qui utilise une phase d'ex-

4. On désigne par là une déclaration de base du code source, typiquement terminée par un point-virgule en `C`, qu'on traduit malheureusement par *instruction* en Français, ce qui crée la confusion avec les *instructions* assembleurs. On utilise donc le terme anglais pour davantage de précision.

périmentation sur carte pour construire un modèle de faute probabiliste en fonction de l'équipement d'attaque et du composant attaqué. Au cœur de cette méthode, la notion de modèle de faute probabiliste, un formalisme pour décrire les types de fautes qui peuvent se produire avec une certaine probabilité en utilisant un équipement d'attaque spécifique sur un composant spécifique. Dans la suite de ce chapitre, on définit la notion de modèle de faute probabiliste, puis on présente notre méthode d'inférence de modèles de faute avant de l'appliquer en détail sur un exemple. Enfin, on donne les résultats obtenus en appliquant cette méthode sur divers composants et avec différents équipements.

3.2. Modèle de faute probabiliste

L'objectif de cette section est de construire la notion de modèle de faute probabiliste de façon à pouvoir répondre à la question : « Quelle est la probabilité d'obtenir un certain paramètre de code $c \in \mathcal{C}$ si j'attaque avec le paramètre d'équipement $p \in \mathcal{P}$? ». Formellement, on cherche :

$$\Pr(C = c \mid p) \tag{3.1}$$

c'est-à-dire, la probabilité que la variable aléatoire C, qui désigne l'effet sur le code lors d'une attaque par perturbation, vaille précisément le paramètre de code c, sachant qu'on attaque avec le paramètre d'équipement p.

Cette construction doit permettre de décrire les fautes réalisables sur un large éventail de composants avec les équipements disponibles. Pour ce faire, on commence par préciser la notion de faute, dans le contexte de l'inférence de modèles de faute.

3.2.1. Fautes

Comme on a pu le voir au début de ce chapitre, les fautes peuvent se décrire suivant deux points de vue différents. D'une part, pour les tests de pénétration, la faute est une modification constatée entre un comportement « normal » (dit nominal) et un comportement observé suite à une attaque par perturbation. D'autre part, du point de vue de la revue de code, on la décrit comme une modification de l'état ou du comportement du composant à un instant temporel et un emplacement spatial précis. L'objectif de notre définition de faute est d'unifier ces deux représentations, en liant l'espace de paramètres de l'équipement d'attaque et l'espace de paramètres du code.

Définition 3.3 *Faute*
Une relation entre l'espace de l'équipement d'attaque et l'espace de paramètres du code. Formellement, une faute est la donnée d'un couple (p, c) avec $p \in \mathcal{P}$ et $c \in \mathcal{C}$.

Notation. Pour exprimer la faute $f \hat{=} (p, c)$, on pourra utiliser la notation $p \underset{f}{\rightsquigarrow} c$. Cette notation exprime que les effets observés dans le domaine du code (paramètres c) sont la conséquence des paramètres d'équipement p choisis.

Exemple 3.1
Avec notre notation, la faute f_{A} qui écrit 0 dans le registre A à l'exécution de l'instruction 124 suite au coup de laser de paramètres $p \hat{=} (x = 12\,\mu\mathrm{m},\ y = 24\,\mu\mathrm{m},\ d = 3800\,\mathrm{ns},\ w = 850\,\mathrm{ns})$ s'écrit :

$$(x = 12\,\mu\mathrm{m},\ y = 24\,\mu\mathrm{m},\ d = 3800\,\mathrm{ns},\ w = 850\,\mathrm{ns}) \underset{f_{\mathrm{A}}}{\rightsquigarrow} (i = 124,\ \mathrm{store}(\mathrm{A},\ 0))$$

Exemple 3.2
La faute f'_{A} volatile de mêmes paramètres s'écrirait :

$$(x = 12\,\mu\mathrm{m},\ y = 24\,\mu\mathrm{m},\ d = 3800\,\mathrm{ns},\ w = 850\,\mathrm{ns}) \underset{f'_{\mathrm{A}}}{\rightsquigarrow} (i = 124,\ \mathrm{read}_m(\mathrm{A},\ 42,\ 0))$$

Cette faute renvoie 0 au lieu de 42 uniquement lors de la lecture du registre A à l'instruction 124.

3.2.2. Modèles de faute

On s'intéresse maintenant non plus à une seule faute, mais à des ensembles de fautes. Dans le cadre de notre formalisation, on définit un modèle de faute comme un ensemble de fautes.

Définition 3.4 *Modèle de faute*
Un modèle de faute est un sous-ensemble M de $\mathcal{P} \times \mathcal{C}$.

Exemple 3.3
On donne un exemple de modèle de faute auquel appartient entre autres la faute f_{A} de l'exemple 3.1 :

$$\{(x = 12,\ y = 24,\ d = 3000 + 200k,\ w = 850) \underset{f_{\mathrm{A}}(k)}{\rightsquigarrow} (i = 120 + k,\ \mathrm{store}(\mathrm{A},\ 0)),\ k \in \mathbb{N}\}$$

Dans cet exemple, on relie le délai d'attaque $(3000 + 200k)$ au numéro d'instruction attaqué $(120 + k)$. On a bien $f_{\mathrm{A}} \in M$ pour $k = 4$.

3.2.3. Modèles de fautes probabilistes

En ayant défini une faute et un modèle de faute, la définition d'un modèle de faute probabiliste découle naturellement.

Définition 3.5 *Modèle de faute probabiliste.*
Un modèle de faute probabiliste $\mathcal{M}_{d,e}$ lié au composant d et à l'équipement d'attaque e est une liste de couples $(M, \Pr(M))$ qui sont chacun constitués d'un ensemble de fautes M et de la probabilité d'occurrence $\Pr(M)$ associée à cet ensemble, dans l'intervalle $]0, 1]$.

On souhaite montrer que cette définition permet de calculer $\Pr(C = c \mid p)$. On fournit donc une formule pour le faire :

Calcul de $\Pr(C = c \mid p)$. Soient $c \in \mathcal{C}$, $p \in \mathcal{P}$ et $\mathcal{M}_{d,e}$ un modèle de faute probabiliste constitués de n ensembles de fautes $\{M_0, \ldots, M_{n-1}\}$ et des probabilités associées $\Pr(M_0), \ldots, \Pr(M_{n-1})$. On a alors :

$$\Pr(C = c \mid p) = \sum_{i \in \{0,\ldots,n-1\}} (\Pr(M_i),\ (p, c) \in M_i)$$

c'est-à-dire que la probabilité d'obtenir un certain effet c à partir d'une perturbation p est la somme des probabilités d'obtenir cet effet de chaque ensemble de fautes auxquels la faute $p \rightsquigarrow c$ appartient. On note en particulier que si c n'appartient pas à l'espace de paramètres de code décrit par $\mathcal{M}_{d,e}$, alors $\Pr(C = c \mid p) = 0$, quel que soit p. Cette propriété permet d'ignorer durant la revue de code les effets sur le code qui n'ont aucune probabilité de se produire.

Dans la suite de cette section, on donne deux exemples de modèles de fautes probabilistes : un modèle de faute « idéal » et un modèle de faute pour les fautes multiples indépendantes.

3.2.3.1. Un modèle de faute idéal

On décrit ce modèle de faute comme « idéal » car c'est l'archétype de ce qu'on pourrait souhaiter obtenir en tant qu'attaquant, par exemple pour réaliser l'attaque BellCore (voir page 18). On notera que, comme la plupart de ce qu'on qualifie d'« idéal », on n'a pas rencontré ce modèle de faute aussi clairement délimité dans la réalité des expériences sur carte.

Ce modèle de faute probabiliste décrit des modifications de lectures RAM via des *bitset* et des *bitreset* suite à des perturbations laser. Le nombre de bits écrasés dépend de la durée w d'éclairement. La valeur écrasée (*bitset* ou *bitreset*) dépend de la position (x, y) du laser.

Concrètement, il existe un temps initial t_0, une durée w_0 d'éclairement et 2 positions (x_{00}, y_{00}) et $(x_{\mathrm{FF}}, y_{\mathrm{FF}})$ qui injectent respectivement des *bitreset* et des *bitset*. On définit les paramètres qui injectent des *bitreset* :

$$\forall n, m \in \mathbb{N},\ m \neq 0,\ p_{00}(n, m) \hat{=} (d = t_0 + w_0 \times n,\ w = w_0 \times m,\ x = x_{00},\ y = y_{00}),$$

les paramètres qui injectent des *bitset* :

$$\forall n, m \in \mathbb{N},\ m \neq 0,\ p_{\mathrm{FF}}(n, m) \hat{=} (d = t_0 + w_0 \times n,\ w = w_0 \times m,\ x = x_{\mathrm{FF}},\ y = y_{\mathrm{FF}}),$$

Faute	Probabilité
$f_{00}(n, m))$	1
$f_{\mathrm{FF}}(n, m)$	1

TABLE 3.1. – Modèle de faute probabiliste idéal

la modification correspondant à un *bitreset* :

$$c_{00}(k, j) \hat{=} (i = k,\ \mathrm{read}_m(RAM,\ a,\ a\ \&\ (\texttt{0xFF} - 2^j))),$$

et la modification correspondant à un *bitset* :

$$c_{\mathrm{FF}}(k, j) \hat{=} (i = k,\ \mathrm{read}_m(RAM,\ a,\ a \mid 2^j)).$$

On a alors les ensembles de fautes :

$$\{\forall n, m \in \mathbb{N}, m \neq 0, p_{00}(n, m) \underset{f_{00}}{\rightsquigarrow} \Big(c_{00}(n/8, n \bmod 8), c_{00}((n+1)/8, (n+1) \bmod 8), \ldots, c_{00}((n+m-1)/8, (n+m-1) \bmod 8) \Big) \}$$

et

$$\{\forall n, m \in \mathbb{N}, m \neq 0, p_{\mathrm{FF}}(n, m) \underset{f_{\mathrm{FF}}}{\rightsquigarrow} \Big(c_{\mathrm{FF}}(n/8, n \bmod 8), c_{\mathrm{FF}}((n+1)/8, (n+1) \bmod 8), \ldots, c_{\mathrm{FF}}((n+m-1)/8, (n+m-1) \bmod 8) \Big) \}$$

Dans ce modèle idéal, à chaque attaque suivant un paramètre de la forme p_{00}, l'attaquant obtient une faute de l'ensemble f_{00}, et à chaque attaque suivant un paramètre de la forme p_{FF}, l'attaquant obtient une faute de la forme f_{FF}. On obtient donc le modèle de faute probabiliste décrit dans le tableau 3.1. On note que la somme des probabilités est supérieure à 1 : la raison est que les paramètres de la forme p_{00} et p_{FF} sont disjoints ($x_{00} \neq x_{\mathrm{FF}}$ et $y_{00} \neq y_{\mathrm{FF}}$). Pour les modèles où un même paramètre p peut conduire à différents résultats, la somme des probabilités de ces différents résultats doit valoir 1.

Dans la pratique, nous n'avons pas rencontré de modèles aussi bien délimités. Dans les modèles concrètement observés, on note les différences suivantes :

— Un même paramètre p donne des résultats variables, à savoir des cas nominaux, des mutismes de la carte, voire différents types de fautes.
— On n'a pas observé de modèle permettant de n'affecter qu'un seul bit d'un emplacement mémoire. En général, on modifie au moins un octet, même s'il existe des cas limites plus ou moins courants où on modifie seulement une partie d'un octet.
— On s'est concentré sur les fautes concernant la mémoire non volatile (type mémoire flash) et non la RAM. Les modèles qu'on décrit dans la suite ne concernent donc pas la mémoire RAM.

On pourra constater toutes ces différences dans la section 3.4 qui propose une étude de cas concrète d'inférence de modèle de faute.

Faute	Probabilité
$f_{00}(n,m), f_{\mathrm{FF}}(j,k)$	1

TABLE 3.2. – Modèle de faute probabiliste pour des fautes multiples indépendantes

3.2.3.2. Le cas des fautes multiples indépendantes

Supposons qu'on dispose d'un composant et d'un laser nous permettant d'obtenir le modèle de faute idéal décrit au paragraphe précédent. On se munit d'un second laser de mêmes caractéristiques que le premier pour effectuer une seconde perturbation indépendante de la première.

Chacun des lasers étant munis du modèle de faute idéal, les paramètres p_{00} et p_{FF} peuvent être choisis pour chacun des deux lasers. Dans la section précédente, on a présentés x_{00}, x_{FF}, y_{00} et y_{FF} comme étant des valeurs uniques. Dans ce cas, les deux lasers ne pouvant pas simultanément occuper la même position, l'un des deux lasers est positionné à un paramètre de la forme p_{00} et l'autre à un paramètre de la forme p_{FF}. Un paramètre p d'équipement concernant les deux lasers est donc l'union de deux paramètres p_{00} et p_{FF}. On ne spécifie pas ce qui se produit si la forme des paramètres induit à la fois des *bitreset* et des *bitset* sur les mêmes bits d'une lecture mémoire, c'est-à-dire ce qu'il se passe si les deux lasers attaquent « en même temps ».

On obtient alors le modèle de faute décrit dans le tableau 3.2. On notera que dans ce tableau il faut que les intervalles $[n,\ldots,n+m[$ et $[j,\ldots,j+k[$ soient disjoints. On considère que la probabilité d'occurrence est toujours 1. Ce résultat peut paraître intuitif : chaque coup de laser produit la faute associée de façon certaine, donc deux coups de laser produisent les deux fautes de façon certaine. Ce raisonnement fait l'hypothèse implicite que les deux coups de laser sont indépendants. Cette hypothèse est difficile à établir dans le cas général (par exemple, on se doute qu'elle est fausse dans les cas où les deux lasers tirent en même temps). Dans la pratique, si les deux coups ont lieu à des instants différents, elle est généralement vérifiée, aux problèmes de synchronisation près (il peut être techniquement plus difficile de synchroniser deux coups de lasers qu'un seul). On calcule alors la probabilité de réussir les deux fautes comme le produit des probabilités de réussir chaque faute séparément (ici, 1×1 nous donne 1).

3.3. La méthode d'inférence de modèles de faute

La méthode présentée dans cette section est une généralisation de celle présentée à la conférence CARDIS 2015 [Dur+15]. On commence par présenter l'état de l'art des méthodes d'inférence et les difficultés rencontrées par ces méthodes. Ensuite, on définit et on donne plusieurs exemples d'un type de programme particulier, qu'on appellera « programme de détection de fautes », et qui joue un rôle clé dans la méthode que nous proposons. Enfin, on détaille les trois phases de la méthode d'inférence de modèles de faute proprement dite.

3.3.1. État de l'art des méthodes d'inférence et difficultés

3.3.1.1. Validation du modèle de faute NOP

Moro *et al.* publient en 2013 une validation du modèle de faute NOP [Mor+13]. Pour la réaliser, les auteurs ont effectué des expériences d'attaque par perturbation sur carte puis collecté tous les comportements observés. Ensuite, ils ont simulé les fautes suivant deux modèles : le remplacement d'instruction et le saut d'instruction (modèle dit « NOP »), l'objectif étant de lister les fautes conduisant aux comportements observés sur carte. Les auteurs ont déterminé que la simulation exhaustive de remplacement d'instruction était trop coûteuse en temps pour cette tâche, et ont donc utilisé seulement les résultats du modèle de saut d'instruction. Ces résultats leur ont permis de déterminer que le modèle de saut d'instruction couvre 25% de tous les comportements observés sur carte. Cette approche de validation d'un modèle très utilisé pour modéliser les fautes (modèle de saut d'instruction) est intéressante, ainsi que l'analyse que Moro *et al.* en font : dans la pratique sur carte, il est courant que la faute ne corresponde pas vraiment à un saut d'instruction, mais plutôt à un remplacement d'instruction, toutefois celui-ci est sans effet fonctionnel dans le programme considéré (par exemple, il met à jour un registre, mais cette valeur fautée n'est pas utilisée par le programme). Le modèle de saut d'instruction peut alors être vu comme la suppression des effets de bords d'une partie du programme initial. Cette étude de l'impact « fonctionnel » des fautes donne donc une représentativité supérieure au modèle de saut d'instruction. Cependant, elle devient alors très dépendante du programme considéré. En effet, deux programmes différents peuvent ou non utiliser un registre fauté. Cette approche a été menée sur un unique composant non sécurisé sur une base Cortex-M3 (ARMv7-M). Il serait intéressant de constater la variation de la couverture des fautes par le modèle de saut d'instruction sur des composants non ARM et des composants sécurisés.

3.3.1.2. Extraction de modèles de faute

Rivière *et al.* proposent en 2015 une approche d'extraction de modèles de faute appliquée au cache d'un composant non sécurisé sur une base Cortex-M3 [Riv+15]. Leur approche est d'utiliser des instructions non-idempotentes [5] pour déterminer précisément le point d'injection de la faute pendant des expériences d'attaques par perturbation sur carte. Les auteurs constatent qu'en attaquant dans la zone spatiale correspondant au cache d'instruction de la carte, ils peuvent modifier les instructions exécutées : l'exécution de plusieurs instructions est remplacée par la réexécution d'instructions précédemment dans le cache. L'approche présentée est intéressante, car elle permet d'étudier l'impact des différents paramètres de l'équipement sur la probabilité d'obtenir le comportement observé. En revanche, l'approche semble très « manuelle » et très spécialisée pour ce composant. Enfin, la méthode utilisée pour extraire la faute réalisée à partir des comportements observés sur carte n'est pas totalement claire.

5. Deux instructions sont dites idempotentes lorsque exécuter les deux instructions a le même effet qu'exécuter une seule des deux. Par exemple, une addition de deux registres dans un troisième constitue une addition idempotente. À l'inverse, deux instructions sont non-idempotentes lorsque exécuter une seule des deux n'a pas le même effet qu'exécuter les deux.

3.3.1.3. Difficultés à lever

Toutes les méthodes d'inférence de modèles de faute doivent faire face à trois difficultés : (1) le problème de la taille de l'espace des paramètres, (2) l'influence du composant et de l'équipement, et (3) l'influence du programme considéré.

1. Le problème de la taille de l'espace des paramètres. L'espace des paramètres d'équipement est trop grand pour pouvoir être testé exhaustivement. En effet, pour une carte de 3.5 mm de côté testé au laser avec une précision de déplacement de 1 µm, il y a $3500 \times 3500 = 12\,250\,000$ positions spatiales possibles. Pour **chacune** de ces positions, pour un code de 2 µs, avec une précision de 20 ns, il y 100 délais possibles à tester. De même, la durée d'éclairement peut se tester de 200 ns à 2000 ns par pas de 50 ns, soit 36 valeurs. Pour chacune de ces $12\,250\,000 \times 100 \times 36 = 441 \times 10^8$ positions spatio-temporelles, on peut aussi faire varier l'intensité d'éclairement. Même en se limitant à 2 valeurs (une intensité faible et une intensité élevée), on a déjà 882×10^8 valeurs possibles. En supposant qu'on peut effectuer 10 attaques par seconde (une nombre assez optimiste), il faut donc 882×10^7 secondes pour tester une fois toutes les valeurs des paramètres. Ceci représente environ 280 années de tests ininterrompus ! Dans la pratique, il est donc impossible de tester toutes les valeurs des paramètres, c'est pourquoi les évaluateurs cherchent empiriquement des valeurs de paramètres pour lesquelles l'injection de fautes semble efficace.

2. L'influence du composant et de l'équipement. Les divers composants utilisent des architectures matérielles différentes (processeurs, technologies de gravure, de mémoire non volatile, etc.). Empiriquement, les évaluateurs CESTI ont déterminé que le modèle de faute accessible dépend ainsi du composant considéré (certaines mémoires n'autorisent que l'écrasement à 0, d'autres à FF, d'autres encore à des valeurs qui semblent aléatoires). De même, les différents équipements, bien qu'ayant pour conséquence l'injection de fautes, ne garantissent pas de produire les mêmes fautes. Il est difficile pour les différentes approches d'inférence de modèles de faute de tenir compte de cette variabilité.

3. L'influence du programme considéré. Moro *et al.* valident le modèle de faute NOP au niveau fonctionnel, c'est-à-dire en comparant leur effet dans le contexte applicatif. Cette méthode pose cependant un problème : deux fautes différentes, mais indistinguables dans le contexte applicatif testé, seront considérées comme identiques. Dans le cas général, il est cependant très difficile de distinguer tous les types de fautes, car sur carte on ne peut observer que l'état final de la mémoire et des registres après l'attaque par perturbation. Supposons qu'on observe ainsi qu'un registre A a une valeur différente de celle attendue. Cette différence peut provenir d'au moins trois causes différentes : une corruption directe du registre A et donc de sa valeur, une corruption de la valeur mémoire précédemment chargée dans A, ou le remplacement d'une instruction quelconque en un chargement mémoire dans A. Dans le programme considéré, et d'un point de vue fonctionnel, ces trois fautes ont le même effet et peuvent être confondues. Toutefois, si l'on cherche à établir un modèle de faute réutilisable pour différents programmes, ces fautes doivent être différenciées, car leurs effets sont différents dans le cas général.

3.3.2. Les programmes de détection de faute

Pour pallier le problème précédent (influence du programme considéré), on propose les programmes de détection de faute, qui visent à déterminer les causes précises des comportements observés. Ces programmes sont construits afin qu'ils puissent distinguer entre les différentes fautes possibles. Pour reprendre l'exemple du registre, un programme de détection adapté aurait une sémantique différente en fonction de la faute mise en jeu. L'intérêt des programmes de détection est qu'ils peuvent être appliqués en séquence pour obtenir de nouvelles informations sur le modèle de faute en réutilisant les informations obtenues grâce aux programmes de détection précédents. Dans ce contexte, on construira souvent les programmes de détection *ad-hoc* pour valider ou infirmer des hypothèses sur le modèle de faute. Néanmoins, certains programmes peuvent être réadaptés pour fonctionner dans plusieurs contextes.

On propose trois programmes de détection de faute. Le premier (section 3.3.2.1) est un programme de détection des fautes dans la mémoire non volatile, qui est très pratique car il ne nécessite aucune hypothèse préalable. On l'utilise donc souvent dans la méthode d'inférence. Le second programme (section 3.3.2.2) vise à détecter des fautes dans les registres, et requiert une hypothèse d'intégrité de la mémoire non volatile (qui peut être établie à l'aide du premier programme). Enfin, le troisième programme (section 3.3.2.3) donne l'idée d'un programme qui permettrait de déterminer précisément des fautes par remplacement d'instruction. On fournit ce dernier programme pour discuter l'idée, mais il n'est pas applicable en l'état sur des composants réels.

3.3.2.1. Détection des fautes dans la mémoire non volatile

Ce programme de détection de faute a été présenté à [Dur+15]. Son but est de déterminer quelles fautes perturbent la mémoire non volatile (dans la suite, on parlera d'EEPROM, mais le principe serait le même pour d'autres types de mémoires non volatiles), en s'assurant que ces fautes ne perturbent pas la mémoire volatile (RAM) ou les registres. Le listing 3.1 donne l'idée principale de ce programme, écrit pour l'architecture ARMv7-M : initialement, le registre `r0` pointe vers le début d'un tampon EEPROM, `r1` au début d'un tampon RAM, et `r2`, `r3` vers différentes parties du tampon de sortie. Le programme effectue une copie du tampon EEPROM vers le tampon de sortie et une copie du tampon RAM vers le tampon de sortie. Le tampon de sortie se situe lui-même en RAM. Il est important de noter que ce programme doit être placé et exécuté depuis la RAM, et non depuis l'EEPROM. En effet, comme le code est en RAM, si la RAM est intègre après une attaque par perturbation, alors le code est intègre. Dans ce cas, les fautes dans la copie EEPROM ne peuvent résulter que de fautes dans l'EEPROM. En revanche, si la copie du tampon RAM est perturbée, alors on ne peut pas déterminer la nature de la faute avec ce programme. Dans ce contexte, on dit que le tampon RAM sert de sentinelle.

Ce programme de détection se paramètre par les valeurs contenues dans le tampon EEPROM. Dans les expériences, on a choisi systématiquement la même valeur pour toutes les cases du tampon afin de réduire le nombre de cas à tester. Ce choix simplificateur implique cependant qu'on peut passer à côté de certains effets, comme par exemple un « décalage » de la lecture dans le tampon.

Le programme est adaptable à d'autres composants, même d'architectures différentes, tant qu'ils permettent l'exécution depuis la RAM. Il est également possible d'échanger les rôles de l'EEPROM et de la RAM pour établir le modèle de faute sur la RAM (ceci est cependant coûteux en écritures EEPROM dans la pratique).

</>Code assembleur

```
1  ; main_loop:
2    58:        ldrb        r5, [r0, #0] ; r5 <- @EEPROM
3    5a:        strb        r5, [r2, #0] ; r5 -> @IO_EEPROM
4    5c:        ldrb        r5, [r1, #0] ; r5 <- @RAM
5    5e:        strb        r5, [r3, #0] ; r5 -> @IO_RAM
6    60:        add.w        r0, r0, #1 ; @EEPROM += 1
7    64:        add.w        r1, r1, #1 ; @RAM += 1
8    68:        add.w        r2, r2, #1 ; @IO_EEPROM += 1
9    6c:        add.w        r3, r3, #1 ; @IO_RAM += 1
```

Listing 3.1. – Programme de détection de fautes dans la mémoire non volatile

3.3.2.2. Détection des fautes dans les registres

Ce programme vise à déterminer les fautes susceptibles de corrompre directement la valeur d'un registre. Il repose sur l'hypothèse que le code est intègre. L'idéal est donc d'exécuter ce programme depuis l'EEPROM après avoir caractérisé les fautes sur l'EEPROM en utilisant le programme de la section précédente. Ceci permet de se placer à des valeurs p telles que l'EEPROM n'est pas perturbée.

Le listing 3.2 est une implémentation ARMv7-M. Le programme se paramètre par le registre visé `rn`, et est partagé en trois régions :

1. Une phase d'initialisation où une constante est assignée à `rn`
2. Une phase sentinelle où le contenu du registre `rn` est écrit dans le tampon EEPROM de sortie.
3. Une phase de plusieurs écritures successives du registre `rn` dans le tampon EEPROM de sortie.

Chacune des régions est séparée des autres par de nombreuses instructions `nop` (qui n'ont aucun effet sur l'exécution) afin de faciliter l'étude temporelle. La phase sentinelle sert à garantir que la phase d'initialisation a bien lieu : si la valeur écrite pendant la phase sentinelle est différente de la constante de la phase d'initialisation, les résultats sont invalides. Si cette valeur est intègre, le tampon EEPROM de sortie peut être lu pour constater si la valeur du registre `rn` a été perturbée. Les positions et le nombre de valeurs perturbées donnent l'instant temporel (dans l'espace du code) où la faute a débuté et sa durée (pour des attaques sur registres, on s'attend en général à ce que la faute soit permanente, donc jusqu'à la fin du tampon).

Pour épargner un nombre massif d'écritures EEPROM, on peut ajouter une hypothèse supplémentaire d'intégrité de la RAM pour remplacer les écritures EEPROM par des écritures RAM.

</>Code assembleur

```
; initialisation
  mov         rn, #x ; rn <- x
  nop
  nop
  ...
  nop
  nop
; sentinel
  str   rn, [rm, #0] ; rm contains @IO_EEPROM
  nop
  nop
  ...
  nop
  nop
; writes
  str   rn, [rm, #4] ; IO_EEPROM[4] <- x
  str   rn, [rm, #8] ; IO_EEPROM[8] <- x
  ...
```

Listing 3.2. – Programme de détection de fautes dans les registres

3.3.2.3. Détection de remplacements d'instructions

On détaille la démarche, sur un jeu d'instructions simplifié, qui permettrait de déterminer un modèle de faute de remplacement d'instructions de façon itérative. On fait les hypothèses que la RAM et les registres sont intègres, seule l'EEPROM est vulnérable, et également l'hypothèse que les valeurs initiales n'ont pas d'influence sur les valeurs perturbées.

Pour illustrer l'approche proposée, on considère un jeu d'instructions simple qui contient un unique registre `r` qu'on suppose préchargé à la valeur 12. On dispose de trois instructions `add r, #x`, `sub r, #x` et `mul r, #x` qui effectuent respectivement l'addition, la soustraction et la multiplication du registre `r` avec la constante `#x` sur 1 octet. Ce résultat est stocké dans `r`, et la constante `#x` est encodée dans l'instruction. On donne l'encodage des différences instructions :

— `add x` : 00101011 xxxxxxxx
— `sub x` : 00101101 xxxxxxxx
— `mul x` : 00101010 xxxxxxxx

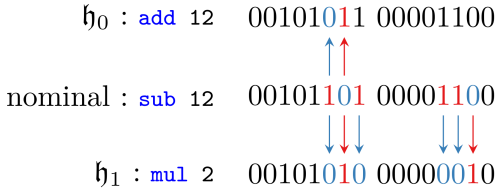
On propose un premier programme constitué d'une unique instruction : `sub 12`. Le résultat nominal de ce programme est donc 0. En attaquant la carte, on observe 24. Les instructions existantes nous donnent les deux hypothèses de remplacement d'instruction suivantes :

— $\mathfrak{h}_0$: `sub 12` → `add 12`
— $\mathfrak{h}_1$: `sub 12` → `mul 2`

L'encodage des trois instructions considérées est alors le suivant :

— `sub 12` (nominal) : 00101101 00001100
— `add 12` ($\mathfrak{h}_0$) : 00101011 00001100
— `mul 2` ($\mathfrak{h}_1$) : 00101010 00000010

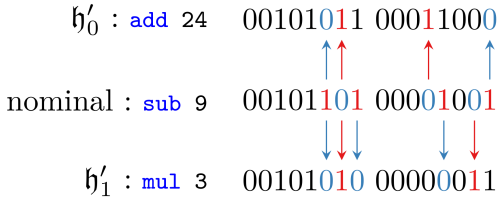
Si on exprime les deux hypothèses de remplacement comme une série de *bitresets* et de *bitsets* sur les encodages à partir de l'encodage initial, on obtient donc :



Pour déterminer l'hypothèse correcte entre $\mathfrak{h}_0$ et $\mathfrak{h}_1$, on propose un deuxième programme constitué de l'instruction `sub 9` (encodage 00101101 00001001). La valeur nominale est donc 3. De nouvelles perturbations permettent d'observer la valeur 36. Les instructions possibles nous donnent les deux nouvelles hypothèses de remplacement :

— $\mathfrak{h}'_0$: `sub 9` → `add 24`
— $\mathfrak{h}'_1$: `sub 9` → `mul 3`

Ou, du point de vue des encodages :



Ces modifications d'encodage permettent de départager $\mathfrak{h}_0$ et $\mathfrak{h}_1$. En effet, la présence du *bitset* du bit 4 du mot 2 pour pouvoir remplacer `sub 9` par `add 24` est incohérente avec l'hypothèse $\mathfrak{h}_0$ où ce *bitset* n'avait pas lieu. En revanche, les modifications de $\mathfrak{h}'_1$ sont compatibles avec $\mathfrak{h}_1$. Ce second test infirme donc $\mathfrak{h}_0$ et confirme $\mathfrak{h}_1$.

Dans la pratique la complexité est beaucoup plus grande, et ce programme de détection n'est pas utilisable, car il est difficile de cibler précisément une seule instruction (on doit donc émettre des hypothèses sur plusieurs instructions). Par ailleurs, les résultats d'une injection de fautes dépendent aussi de paramètres qu'on ne peut ni contrôler ni mesurer et qui peuvent conduire à des résultats variables pour un paramètre p d'équipement fixé, ce qui a pour effet de beaucoup augmenter le nombre d'hypothèses à tester. Par contre, on pourrait utiliser une technique similaire pour observer l'effet de l'injection de faute sur de petits programmes de test constitués de structures basiques de flot de données (assignations) ou de contrôle (conditions, boucles). Cette approche est suggérée par Rivière dans son manuscrit de thèse [Riv15] et il est prévu de l'expérimenter dans le cadre du projet SERTIF [6].

3.3.3. Vue d'ensemble de la méthode

La figure 3.2 illustre le déroulement de la méthode d'inférence de modèle de faute. L'objectif de la méthode est d'obtenir puis de raffiner un ensemble d'hypothèses sur les fautes qui font partie ou non du modèle de faute probabiliste. Par « hypothèse », on entend « l'ensemble des informations recueillies » sur le modèle de faute, en particulier les fautes qu'il contient et celles qu'il ne contient pas. L'approche débute par une phase manuelle d'initialisation afin de construire un ensemble initial d'hypothèses sur

6. sertif-projet.forge.imag.fr

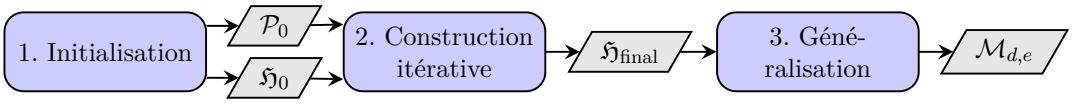


FIGURE 3.2. – Inférence de modèle de fautes

un espace réduit de paramètres d'équipement. Une seconde phase est ensuite répétée de façon itérative afin d'acquérir des données permettant d'éliminer certaines hypothèses sur le modèle de faute, et éventuellement d'en former de nouvelles. Utiliser les hypothèses obtenues lors des itérations précédentes est critique tout au long de cette seconde phase. Une fois que l'ensemble d'hypothèses explique bien les fautes observées, la troisième phase commence. Elle a pour but de généraliser le modèle de faute découvert sur l'espace de paramètres d'équipement initial à un espace de paramètres d'équipement plus grand.

Le modèle de faute probabiliste obtenu est constitué des hypothèses de fautes qui n'ont pas été éliminées par la méthode, auxquelles sont associées une probabilité d'occurrence, déterminée à partir du nombre d'observations de chaque type de faute.

On détaille précisément les trois phases dans la suite de la section.

3.3.3.1. Phase d'initialisation par découverte des paramètres

Au cours de cette phase, qui est manuelle, l'espace des paramètres d'équipement est (partiellement) parcouru pour construire un ensemble initial d'hypothèses. Concrètement, on cherche à construire un ensemble $\mathcal{P}_0 \subset \mathcal{P}$ où des fautes « raisonnablement intéressantes » se produisent « raisonnablement souvent ». Cette phase est rendue nécessaire par le problème de taille de l'espace $\mathcal{P}$ discuté dans l'introduction du chapitre, et elle repose dans la pratique sur l'expertise de l'évaluateur qui applique la méthode en utilisant ses connaissances préalables de son équipement et du composant attaqué. Par exemple, un évaluateur peut généralement déterminer l'aire occupée par la mémoire non volatile à la surface du circuit, et sait empiriquement que les fautes sur la mémoire non volatile se produisent en attaquant aux positions qu'elle occupe physiquement. Ceci permet d'observer, avec un programme de détection de faute simple, les premières fautes qui se produisent. À ce stade, il y a trop peu de données par paramètre pour observer tous les comportements possibles ou pour calculer des probabilités d'occurrence, l'objectif est justement de construire $\mathcal{P}_0$ avec une taille rendant possibles les phases suivantes. À l'issue de cette phase, on a donc un ensemble initial $\mathfrak{H}_0$ d'hypothèses à tester dans $\mathcal{P}_0$.

3.3.3.2. Phase de construction itérative

Cette phase est illustrée sur la figure 3.3. Elle s'effectue sur l'espace réduit $\mathcal{P}_0$ déterminé à la phase d'initialisation, et vise à construire un ensemble d'hypothèses $\mathfrak{H}_{\text{final}}$ présentant idéalement deux propriétés :

1. L'ensemble d'hypothèses couvre (« explique ») toutes les fautes observées via les attaques par perturbation.

2. Les fautes observées par les attaques par perturbation ne sont couvertes que par une unique hypothèse.

La propriété (1) assure que le modèle de faute construit à partir de $\mathfrak{H}_{\text{final}}$ couvre bien les phénomènes observés en leur entier, tandis que la propriété (2) garantit qu'ils sont expliqués de façon non-ambigüe. Ainsi, un ensemble d'hypothèses qui ne vérifie pas (1) laisse certaines fautes inexpliquées, tandis qu'un ensemble d'hypothèses qui ne vérifie pas (2) fournit plusieurs explications « en compétition » pour certaines fautes.

La phase itérative commence avec l'ensemble d'hypothèses $\mathfrak{H}_0$ déterminé durant la phase d'initialisation. À chaque itération, on choisit un programme de détection de faute (étape A), afin de confirmer ou d'infirmer les hypothèses de l'ensemble courant d'hypothèses $\mathfrak{H}_n$. Ensuite, pour chaque paramètre $p \in \mathcal{P}_0$, on effectue plusieurs attaques par perturbation de paramètres p sur le composant exécutant ce programme de détection de fautes (étape B). On fait ensuite un traitement statistique des fautes pour les classer suivant l'effet observé (étape C). On exploite les résultats de ce traitement pour construire $\mathfrak{H}_{n+1}$ (étape D). Si on estime que $\mathfrak{H}_{n+1}$ vérifie les propriétés (1) et (2) (étape E), on peut s'arrêter. Sinon, on fait une nouvelle itération, en adoptant un nouveau programme de détection. L'infirmation d'hypothèses peut être automatisée (il faut tester la présence de comportements contrariant l'hypothèse à infirmer), la validation également (comportements en accord avec l'hypothèse), mais la formulation de nouvelles hypothèses est aujourd'hui un processus manuel, de même que le choix du programme de détection à même de tester les hypothèses. La fréquence de validation des hypothèses permet de déterminer la probabilité d'occurrence de la faute associée.

3.3.3.3. Phase de généralisation

Une fois l'ensemble final d'hypothèses $\mathfrak{H}_{\text{final}}$ et les probabilités associées déterminés, la dernière phase vise à généraliser les résultats obtenus sur $\mathcal{P}_0$ à des espaces de paramètres plus grands. Ceci permet par exemple d'étendre les résultats obtenus en une seule position spatiale à une zone autour de cette position, ou de transposer les résultats obtenus en une position temporelle à d'autres positions temporelles. La généralisation est manuelle. Pour l'effectuer, on a recours à deux techniques :

L'approximation locale. Cette technique part du postulat que deux paramètres d'attaques proches p et p' produisent des fautes proches. Il devient alors possible d'approximer localement autour des paramètres testés, voire d'utiliser des interpolations entre plusieurs paramètres testés. Il est bon de noter que le postulat de localité peut être parfois très faux. Par exemple, les attaques électromagnétiques n'ont pas la même sensibilité spatiale dans toutes les directions : certaines directions sont en effet très peu sensibles au déplacement. D'autres modifient presque immédiatement les résultats (effet « fil »). Néanmoins dans certaines conditions l'approximation locale donne de bons résultats : sous une certaine distance on observe les mêmes fautes, mais avec des probabilités d'occurrence différentes. À partir d'une certaine distance, les fautes observées changent.

Exploitation de la périodicité. Cette technique part du postulat qu'un phénomène qui s'est répété n fois sur la plage de paramètres testés se répétera $n + 1$ sur une plus

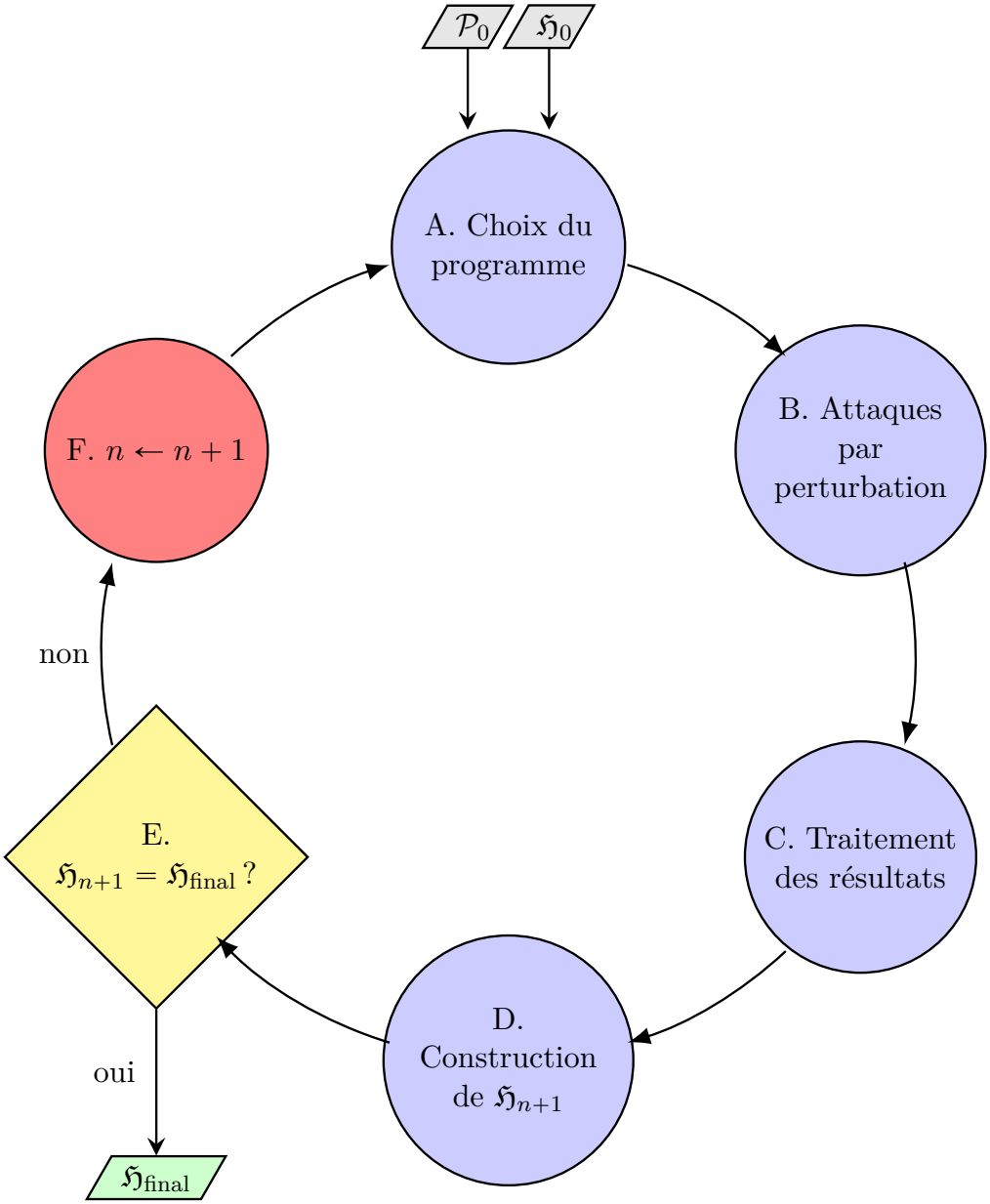


FIGURE 3.3. – Phase de construction itérative

grande plage de paramètres. Elle exploite donc les périodicités observées suivant certaines paramètres. Par exemple, il est courant d'observer une périodicité δ par rapport aux paramètres temporels : autrement dit, attaquer à des délais $d + k\delta$ permet de cibler la $k^{\text{ème}}$ opération. Il s'agit dans cet exemple d'étendre ce résultat à des paramètres temporels $d + k\delta$ qui ne font pas partie de la plage des valeurs temporelles testées. Ce postulat est encore une fois très fort et d'une pertinence très variable. Quand il est utilisé, il est en général associé à une connaissance du composant utilisé (par exemple, la périodicité peut être due aux mécanismes de lecture de la mémoire non volatile qui eux-mêmes se répètent).

Les techniques de généralisation sont utilisées spontanément par les évaluateurs quand ils attaquent des applications : on suppose que les résultats obtenus avec les paramètres testés s'étendent aux valeurs de paramètres voisines. Néanmoins la question est beaucoup moins cruciale pour eux, puisqu'ils cherchent des chemins d'attaque concrets sur l'application et non à généraliser les résultats trouvés. Dans la pratique de l'inférence de modèles de faute, on ne peut pas faire l'économie de la phase de généralisation, car l'espace des paramètres est trop grand pour des tests exhaustifs. Il est important d'au moins citer les techniques de généralisation utilisées avec leurs risques, car elles représentent un point faible de l'approche proposée ici. Pour les cas incertains, un jeu réduit d'expériences peut être effectué pour vérifier expérimentalement le postulat.

3.4. Étude de cas : la carte C

Dans cette section, on présente une étude de cas d'inférence de modèle de faute sur une carte Cortex-M4 (architecture ARMv7-M) non sécurisée qu'on dénotera « carte C » dans la suite. L'objectif de cette étude est d'illustrer comment se déroule la méthode en pratique, et de montrer qu'elle produit bien un modèle de faute probabiliste tel que défini à la section 3.2.3. La carte C est cadencée avec une horloge interne à 8 MHz.[7] On utilise deux types d'équipement :

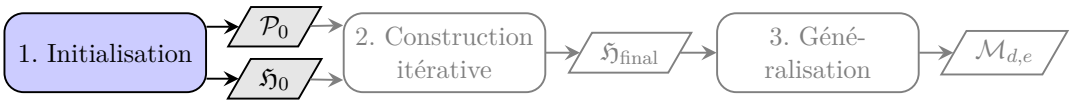
— Un injecteur électromagnétique constitué d'une boucle de cuivre de 100 µm parcourue d'un courant à 500 A pendant 10 ns. **On note $\mathcal{M}_{C,EM}$ le modèle de faute inféré avec cet équipement sur la carte C.**
— Un laser de longueur d'onde $\lambda = 980$ nm et d'une puissance variable. **On note $\mathcal{M}_{C,laser}$ le modèle de faute inféré avec cet équipement sur la carte C.**

On cherche à inférer les modèles de faute sur la flash (mémoire non volatile), car le code des programmes exécutés y est généralement stocké (même s'il est possible d'exécuter du code depuis la RAM).

Dans la suite de cette section, on détaille le processus pour $\mathcal{M}_{C,EM}$ et on donne les résultats pour les deux modèles : la section 3.4.1 correspond à la phase initiale, la section 3.4.2 à la phase itérative et la section 3.4.3 à la phase de généralisation.

7. Toutes les valeurs numériques données dans cette section correspondent aux valeurs utilisées ou observées expérimentalement.

3.4.1. Phase initiale



Dans la phase initiale, on détermine un espace $\mathcal{P}_0$ de paramètres d'équipement pertinent du point de vue de l'injection de fautes, et suffisamment petit pour continuer à dérouler l'approche. Pour ce faire, on étudie séparément l'influence de chaque paramètre sur les fautes produites. Dans cette expérience électromagnétique, les paramètres d'équipement p considérés sont de la forme $p = (\theta, x, y, z, d)$, où θ représente l'angle entre l'injecteur et la carte, (x, y, z) la position de l'injecteur relativement au coin inférieur gauche de la carte exprimée en coordonnées cartésiennes, et d représente le délai avant la décharge électromagnétique. Évaluer l'effet de l'angle θ sur les fautes obtenues est important car l'injecteur émet un champ électromagnétique, disposant d'une direction et d'un sens. On utilise pour cela 4 valeurs d'angles : $-90°$, $0°$, $90°$ et $180°$. La phase initiale se déroule en trois temps :

1. Une cartographie très générale de presque toute la carte à « gros grain », c'est-à-dire en faisant varier les paramètres d'équipement à grands pas, et en faisant peu de répétitions pour chaque valeur des paramètres.
2. Une cartographie plus restreinte de la zone correspondant aux attaques sur la flash.
3. Une étude de l'impact du paramètre z sur les fautes obtenues.

3.4.1.1. Influence de l'angle θ sur la cartographie

On effectue une cartographie très générale d'une grande surface de la carte, en attaquant le programme de détection de fautes non volatiles avec pour valeur d'entrée `0xAA`, avec l'altitude $z = 50\,\mu\text{m}$ fixée, et en faisant varier (x, y), d et θ.

On obtient les 4 figures 3.4a, 3.4b, 3.4c et 3.4d en fonction de la valeur de l'angle θ.[8] Sur ces figures, les points bleus signifient une erreur de la carte (mutisme), les points verts une faute sur la flash en respectant la sentinelle (la sentinelle vient du programme de la section 3.3.2.1 page 70), et les points rouges une faute qui compromet la sentinelle (RAM ou registre). L'absence d'un point peut signifier un comportement nominal ou l'absence de tests en ce point. Pour ces expériences on fait varier x sur la plage $[400\,\mu\text{m}, 2500\,\mu\text{m}]$ et y sur la plage $[500\,\mu\text{m}, 2500\,\mu\text{m}]$, avec pour chacun de ces paramètres un pas spatial de $40\,\mu\text{m}$. On fait varier le paramètre d de $180\,120\,\text{ns}$ à $180\,240\,\text{ns}$ par pas de $40\,\text{ns}$. Pour chaque paramètre p ainsi constitué, on fait 10 expériences, pour un total de $10 \times 4 \times 52 \times 52 \times 3 = 324\,480$ expériences réparties sur 4 jours.

8. Pour des raisons de confidentialité, le fond de carte utilisé sur ces cartographies et les suivantes **ne correspond pas** à une photo du composant attaqué. On a utilisé la photo d'un *die* sous licence libre. Crédit photo : Weitek 3170 die ©2014, by Pauli Rautakorpi (https://commons.wikimedia.org/wiki/User:Birdman86). Licenced under the Creative Commons Attribution 3.0 License, https://creativecommons.org/licenses/by/3.0/

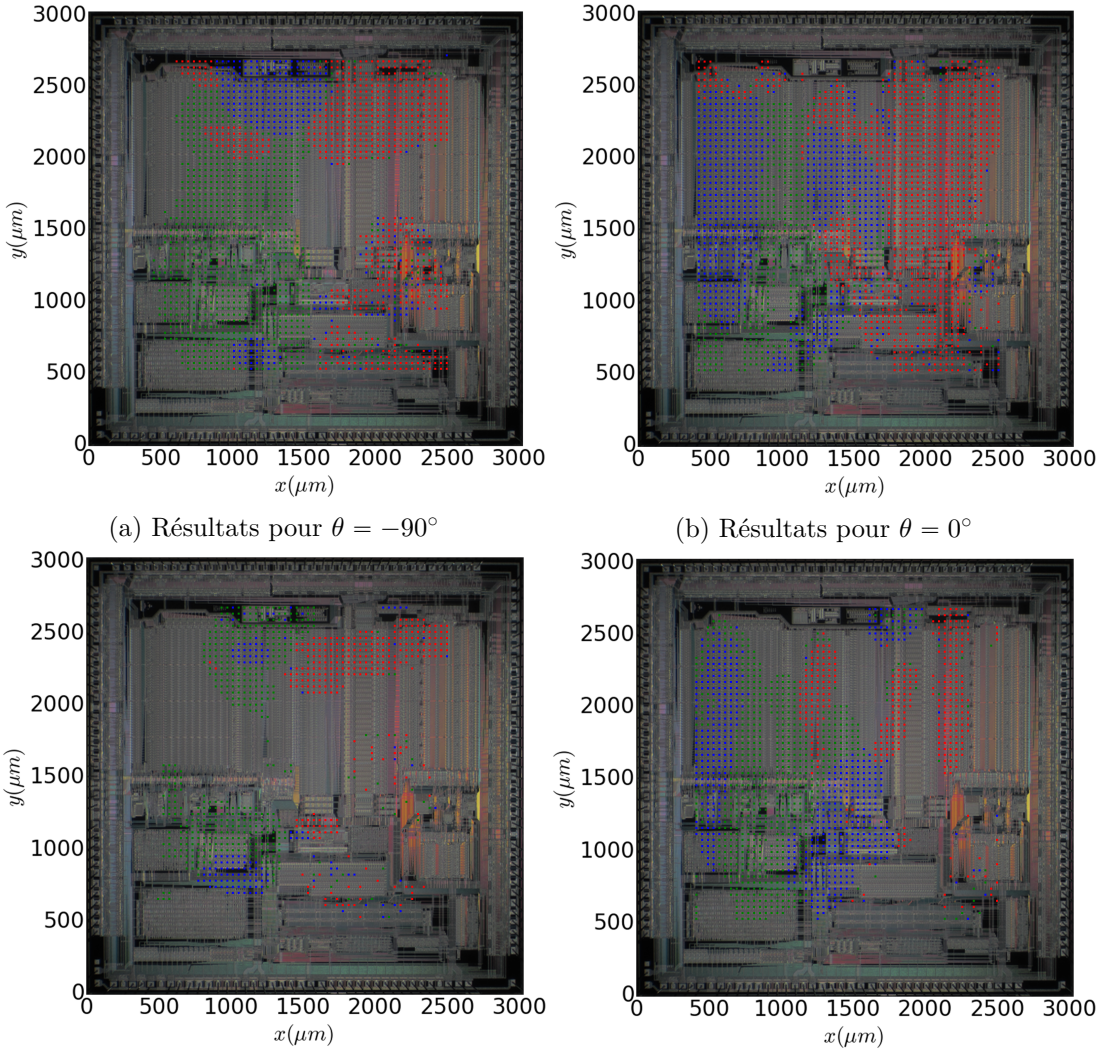
Pour les angles $-90°$ et $90°$, on observe que la nature des fautes injectées varie peu en fonction de x (« lignes » horizontales de points de même couleur). Pour les angles $0°$ et $180°$, on observe de même une faible variabilité en fonction de y (« lignes » verticales de points de la même couleur). Quelle que soit la valeur d'angle, on observe au nord-est des **erreurs RAM** : cette zone correspond à l'emplacement physique de la RAM. Au centre-ouest, on observe des **erreurs flash** : cette zone correspond à l'emplacement physique du contrôleur de la flash. Cette zone est beaucoup plus marquée avec l'angle $-90°$, qu'on va donc privilégier dans la suite du processus. Au centre-ouest nord, avec les angles $-90°$ et $90°$, on observe une zone de mutisme. Avec les angles $180°$ et $0°$, on observe également une zone de mutisme à l'ouest. Contrairement à d'autres expériences (voir section 3.5.1.1) et aux observations dans l'état de l'art [Ord+15], on n'obtient pas de différence significative dans les valeurs injectées en fonction de θ.

3.4.1.2. Cartographie des attaques sur la flash

On a fait une cartographie plus précise des attaques sur la flash, avec des paramètres $p = (\theta = -90°, x = 940\,\mu m, y = 1120\,\mu m, z = 50\,\mu m, d)$, pour $d \in [160\,000\,ns, 200\,000\,ns]$ par pas de 60 ns, et la valeur de test `0xAA` dans le tampon EEPROM du programme de détection de fautes. On fait 10 répétitions par valeur de paramètres, pour un total de $10 \times 666 = 6660$ expériences sur 2 heures. La position (x, y) a été choisie dans la zone de fautes flash. Ces expériences ont permis de constater que les effets obtenus sont des remplacements de 16 octets consécutifs par la valeur 0 dans le tampon flash. On observe de plus que les « blocs » de 16 octets ainsi perturbés sont alignés par 16 octets, c'est-à-dire que l'adresse du premier octet perturbé est congrue à 0 modulo 16. On observe une relation entre le délai d d'attaque et le groupe de 16 octets perturbés. La figure 3.5 en donne la représentation graphique, indiquant la probabilité de fauter le groupe de 16 octets k en fonction du délai d. On y observe que chaque groupe de 16 octets consécutifs est attaqué sur des plages temporelles disjointes de durée 2880 ns, séparées par la durée 1140 ns, pour une périodicité $\delta = 2880 + 1140 = 4020$ ns.

3.4.1.3. Influence de l'altitude z

La figure 3.6 indique l'influence du paramètre z sur les résultats. On constate que z se comporte comme un paramètre énergétique, en influençant la probabilité d'obtenir un comportement fauté, d'erreur (mutisme) ou nominal. Pour des très petites valeurs de z (5 µm), les résultats se partagent exclusivement entre des fautes et des erreurs. Puis, on a une augmentation du taux de faute qui atteint son maximum en $z = 10\,\mu m$ avant de redescendre avec l'augmentation de la distance. Sur cette expérience, le taux d'erreur diminue très rapidement (nul dès $z = 10\,\mu m$) et est remplacé par le taux nominal, qui croît avec la distance. À une grande distance ($z > 150\,\mu m$), l'injecteur n'a plus aucun effet et le taux nominal vaut 1.



(a) Résultats pour $\theta = -90°$

(b) Résultats pour $\theta = 0°$

(c) Résultats pour $\theta = 90°$

(d) Résultats pour $\theta = 180°$

FIGURE 3.4. – Cartographie des fautes pour la carte C en EM

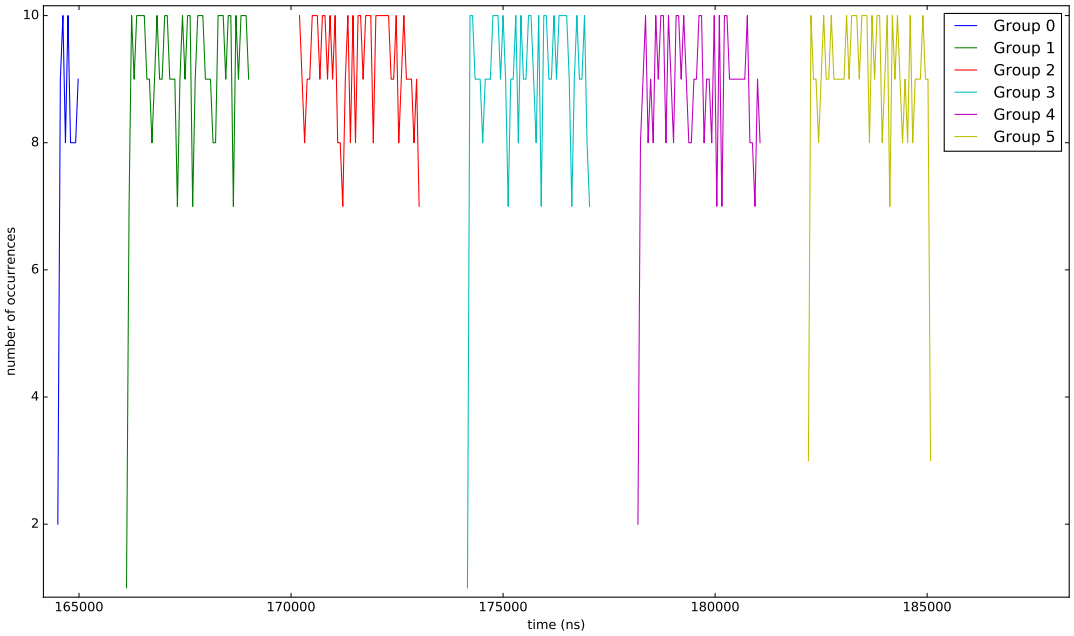


FIGURE 3.5. – Nombre de fautes par groupe de 16 octets en fonction du délai d'attaque

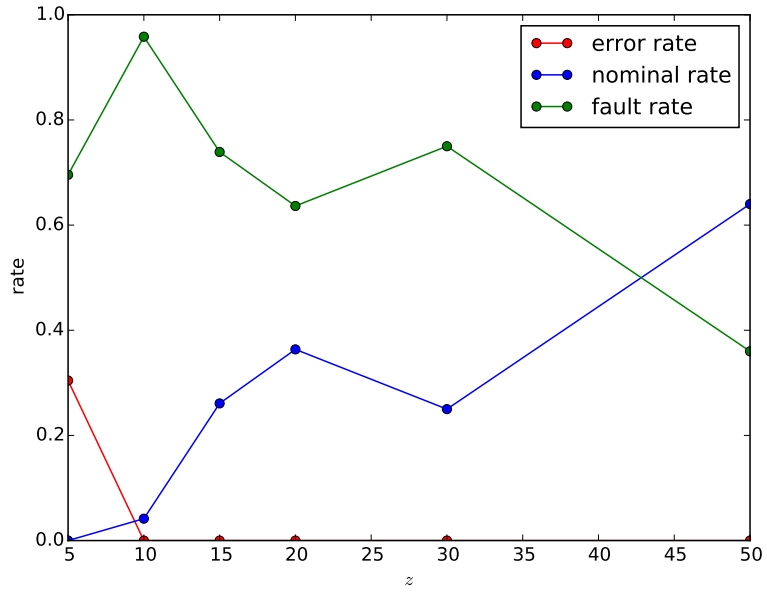
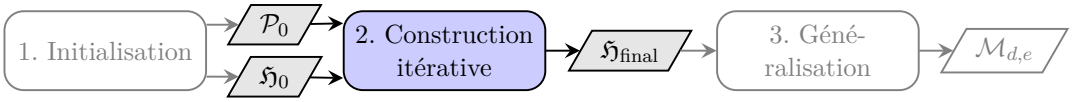


FIGURE 3.6. – Influence du paramètre z

3.4.2. Construction itérative



Dans la construction itérative, on fait une première itération avec le programme de détection de faute dans la mémoire non volatile, qui nous permet de constater un phénomène étrange de « double modèle de faute » et de construire des hypothèses sur les fautes dans la flash. On valide ensuite ces hypothèses au niveau du code exécuté depuis la flash à l'aide d'autres programmes de détection *ad-hoc* décrits à la section 3.4.2.2. On commence donc la construction itérative par l'étude très spécifique d'une petite partie de la zone correspondant aux attaques sur la flash, avec un délai $d_0 = 180\,080\,\text{ns}$ fixé. Les paramètres d'équipement sont ainsi de la forme :

$$p \hat{=} (\theta = -90^\circ, x, y, z = 50\,\mu\text{m}, d = d_0),$$

pour $x \in [780\,\mu\text{m}, 1100\,\mu\text{m}]$ et $y \in [720\,\mu\text{m}, 1680\,\mu\text{m}]$ avec un incrément spatial de $80\,\mu\text{m}$. On utilise le programme de détection de faute sur la mémoire non volatile, en faisant varier la valeur a contenue dans le tampon du programme de détection de faute. On teste ainsi les 256 valeurs possibles d'octets dans un ordre aléatoire avec 20 répétitions pour chaque valeur de paramètres p, pour un total de $20 \times 256 \times 12 \times 4 = 245\,760$ expériences sur 4 jours.

3.4.2.1. Deux modèles de fautes ?

On observe le même type de fautes que précédemment : 16 octets consécutifs mis à `00` à partir d'une adresse alignée. Cependant le détail des expériences révèlent que d'autres valeurs sont parfois écrites à la place de `00`. On cherche à déterminer de quelles valeurs il s'agit, et pour ce faire on trace la figure 3.7, une carte de chaleur qui indique la probabilité que la valeur a du tampon d'entrée soit modifiée en b. Sur cette carte, les points (a, b) qui ne se sont jamais produits sont en noir, les points très peu probables en jaune clair, et les points les plus probables en rouge foncé. On retrouve la très forte probabilité d'obtenir $(a, 0)$ (ligne rouge horizontale), mais on note aussi une faible probabilité d'obtenir `FF`. Il existe également de faibles probabilités d'obtenir des valeurs b inférieures (par *bitreset*) à a, mais elles ne concernent pas équitablement toutes les valeurs a. On a cherché un lien avec les poids de hamming de a, la position, le délai temporel, mais on n'a pas trouvé de corrélation.

On trace la figure 3.8, une carte de chaleur dont l'abscisse n'est pas a, mais l'indice chronologique de a dans notre expérience. En effet, comme on a testé les entrées a entre `00` et `FF` dans un ordre aléatoire, on appelle « indice chronologique » de la valeur a, le numéro qui dénote l'ordre de passage de a dans l'expérience. On constate alors que les valeurs qui ont un comportement irrégulier sont regroupées (sous la forme de « lignes » jaunes verticales) chronologiquement ! Lors du test d'autres programmes, cette modification périodique du comportement se traduit par une diminution subite du nombre de fautes réussies. On a vérifié que ce phénomène était reproductible, mais on n'a pas su déterminer à quoi l'attribuer : les plages horaires où ces effets surviennent semblent aléatoires (ils ne peuvent donc pas être expliqués par l'activité humaine), la température est normalement maintenue à 19 °C, il ne peut pas s'agir d'un « effet mémoire » sur

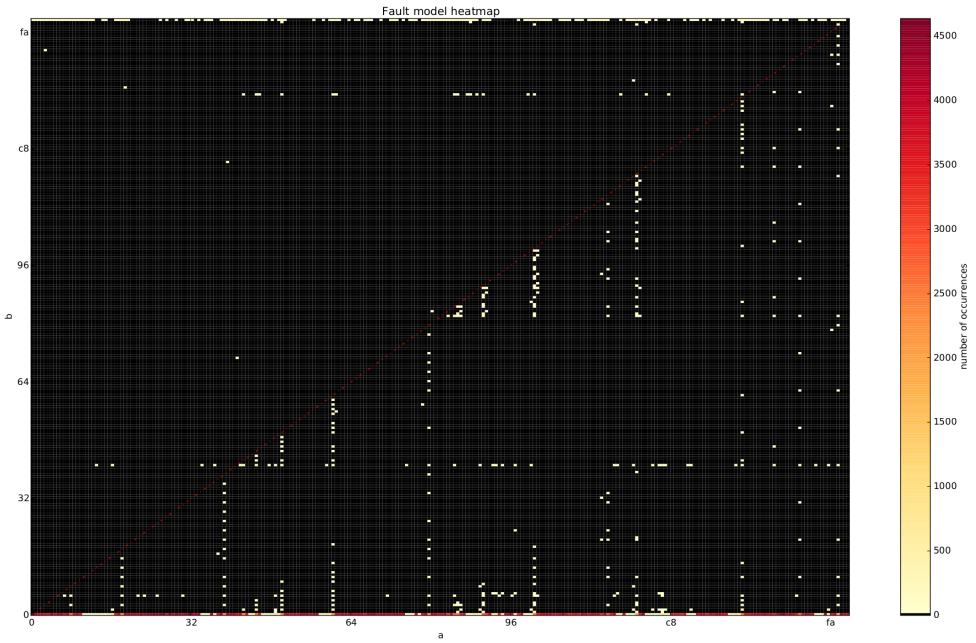


FIGURE 3.7. – Carte de chaleur pour la carte C

des plages d'attaques aussi larges. On peut supposer qu'il y a des changements subtils dans la géométrie de l'injecteur électromagnétique après de nombreuses attaques (qui occasionnent de puissantes décharges à travers la boucle), mais on n'a pas eu les moyens de vérifier cette conjecture. Dans la suite, pour établir le modèle, on exclut les valeurs qui ont un comportement « irrégulier » (on garde donc les 162 valeurs qui présentent un comportement régulier).

On construit alors les deux hypothèses f_{down} et f_{up} :

— $\{(d = d_0 + k\delta) \underset{f_{\text{down}}}{\rightsquigarrow} c_{\text{down}}(k),\ k \in \mathbb{N} \text{ tq } d \in [160\,000, 200\,000]\}$

— $\{(d = d_0 + k\delta) \underset{f_{\text{up}}}{\rightsquigarrow} c_{\text{up}}(k),\ k \in \mathbb{N} \text{ tq } d \in [160\,000, 200\,000]\}$

Dans ces hypothèses, $c_{\text{down}}(k)$ représente la modification du $k^{\text{ème}}$ groupe d'octets de 16 octets pour remplacer leurs valeurs par 00. Soit :

$$\bigcup_{(j\in\{0,\ldots,15\})\wedge(x\equiv 0 \pmod{16})} (i = 16k + j,\ \text{read}_m(EEPROM[x + j],\ a,\ 0))$$

De même, pour $c_{\text{up}}(k)$:

$$\bigcup_{(j\in\{0,\ldots,15\})\wedge(x\equiv 0 \pmod{16})} (i = 16k + j,\ \text{read}_m(EEPROM[x + j],\ a,\ \texttt{FF}))$$

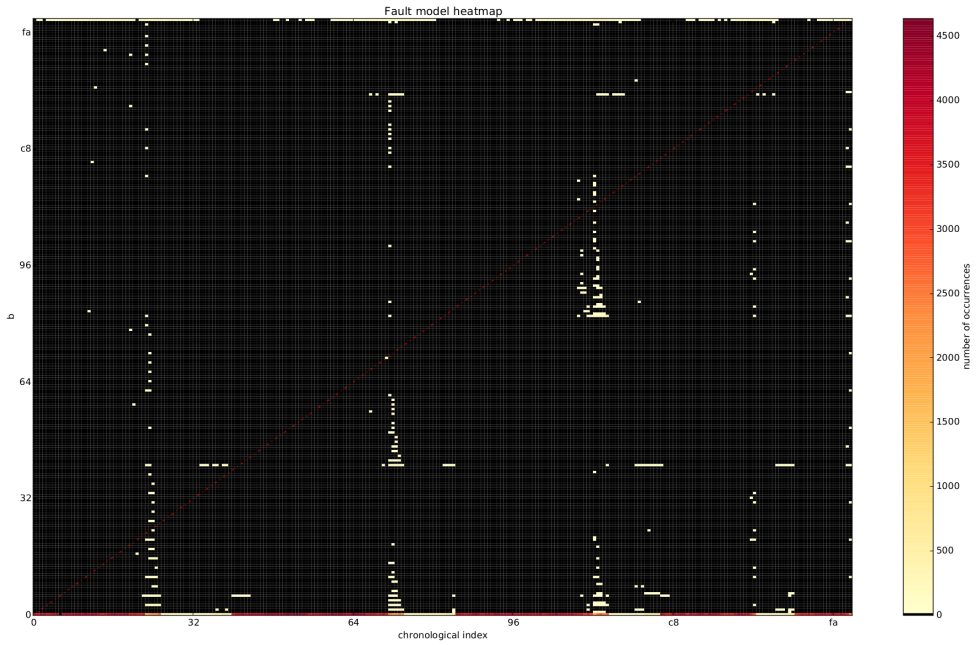


FIGURE 3.8. – Carte de chaleur chronologique pour la carte C

3.4.2.2. Vérification d'hypothèses

L'introduction de 16 `FF` dans du code correspond à des instructions indéfinies en ARMv7-M, et mène au crash de la carte. L'hypothèse f_{up} n'est donc pas vérifiable sur du code. On cherche à vérifier itérativement que l'hypothèse f_{down} s'applique bien sur du code. Pour ce faire, on commence par attaquer un programme de détection constitué essentiellement de `00`. En ARMv7-M, cette instruction correspond à `movs r0, r0`, c'est-à-dire le déplacement de `r0` vers `r0` (équivalent à un NOP, donc), tout en mettant les *flags* d'état à jour si nécessaire. Concrètement, le *flag* Z est mis à `1` si `r0` est nul, et à `0` sinon. **Il ne s'agit donc pas d'un véritable NOP**, car cette mise à jour du *flag* Z peut avoir un effet de bord sur l'exécution du programme. Le programme est constitué d'une étape d'initialisation où des valeurs prédéfinies sont stockées dans les registres `r0` à `r6`, puis d'une suite très longue de `movs r0, r0`, et enfin d'une étape de lecture des registres vers des emplacements prédéfinis de la RAM (on sait que la RAM n'est pas perturbée pour les paramètres p choisis grâce à la phase d'initialisation de l'inférence).

Sur ce premier programme, le but est de vérifier que les fautes n'ont aucun effet particulier sur la valeur des registres. On s'en convainc en attaquant au milieu de l'exécution et en constatant qu'aucune faute n'a lieu (par contre, des mutismes surviennent).

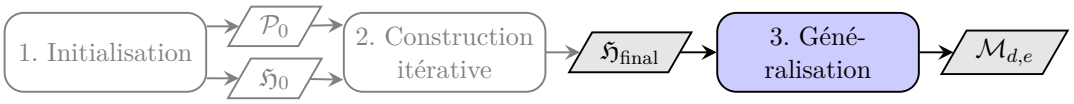
On cherche à vérifier l'hypothèse d'alignement, selon laquelle l'effet sur le code commence à une mémoire d'adresse congrue à 0 modulo 16, en observant le nombre d'instructions remplacées par `movs r0, r0`. On modifie un peu le programme précédent pour insérer une instruction qui modifie `r1` au milieu des `movs r0, r0`. On observe bien

Faute	Probabilité
f_{down}	16%
f_{up}	0.3%

TABLE 3.3. – Modèle de faute probabiliste pour la carte C en EM

qu'avec un certain délai, on arrive à supprimer la modification de `r1`. On augmente la taille et on constate qu'on peut ainsi supprimer jusqu'à 8 instructions consécutives alignées (par exemple entre les adresses `0x4000` et `0x400f`). En revanche, on ne supprime pas 2 instructions consécutives si elles ne sont pas alignées (par exemple aux adresses `0x400e` et `0x4010`). On conclut à la validation de l'hypothèse d'alignement. Ceci achève la phase itérative.

3.4.3. Phase de généralisation



On généralise le comportement des 162 valeurs régulières aux 93 valeurs irrégulières. On justifie cette généralisation par le fait que les valeurs irrégulières ne semblent avoir en commun que leur position chronologique au cours de notre expérience, ce qui semble exclure un comportement spécifiquement causé par les valeurs en question.

On étend la petite zone (x, y) sur laquelle on a fait la construction itérative à l'ensemble de la zone de fautes sur la flash de la figure 3.4a. Cependant, dans la pratique, la probabilité associée aux ensembles qui constituent le modèle varie du simple au double en fonction de la position dans la zone.

Enfin, on exploite la périodicité relevée précédemment pour étendre les résultats depuis la plage $[160\,000\,\text{ns}, 200\,000\,\text{ns}]$ à des intervalles de la forme : $[\delta k, \delta(k+1)]$, pour $k \in \mathbb{N}$ et $\delta = 0\,\text{ns}$ déterminé précédemment.

On donne le modèle final obtenu dans le tableau 3.3.

3.4.4. Modèle laser

Au laser, on retrouve un modèle où l'on perturbe 16 octets consécutifs. La valeur injectée est déterminée par la position où l'on attaque. La figure 3.9 est une cartographie des zones où la mémoire EEPROM est perturbée. En or, les points où la valeur injectée est `00`. En vert, les points où la valeur injectée est `FF`. On note $p_{00}(k)$ les paramètres où l'on injecte `00` avec le délai $d = d_0 + k\delta$, et $p_{\text{ff}}(k)$ les paramètres où l'on injecte `FF` avec le même délai. On a alors les deux ensembles de fautes :

$$\{p_{00}(k) \underset{f_{00}}{\rightsquigarrow} c_{\text{down}}(k)\}$$

et

$$\{p_{\text{ff}}(k) \underset{f_{\text{FF}}}{\rightsquigarrow} c_{\text{up}}(k)\}$$

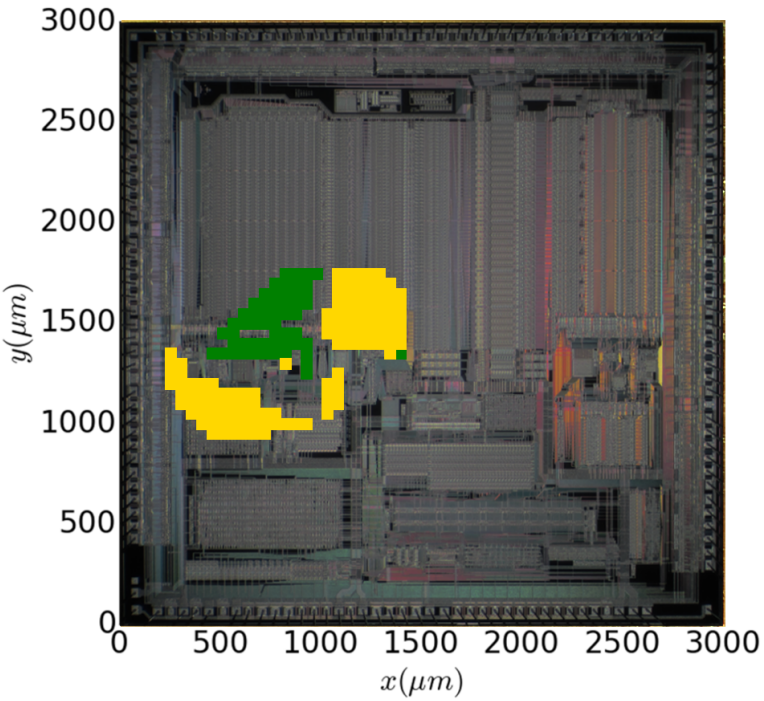


FIGURE 3.9. – Cartographie pour la carte C au laser

Faute	Probabilité
f_{00}	21%
f_{FF}	69%

TABLE 3.4. – Modèle de faute probabiliste pour la carte C en laser

où c_{down} et c_{up} désignent les mêmes ensembles de paramètres de code que pour le modèle électromagnétique. En calculant les probabilités d'occurrence de chaque ensemble de fautes, on obtient le modèle donné à la table 3.4.

Par rapport au modèle électromagnétique, on a des probabilités supérieures de succès dans le modèle laser (mais les effets sur le code de f_{FF} restent peu intéressants), et davantage de contrôle sur la faute à effectuer, puisque les paramètres p_{00} et p_{FF} sont disjoints.

3.5. Conclusion

Dans ce chapitre, on a présenté et mis en œuvre une nouvelle méthode pour inférer un modèle de faute à partir d'expériences sur carte et de programmes spécifiquement conçus pour permettre de déterminer la nature précise des fautes injectées. Le modèle de faute obtenu est lié à l'équipement d'attaque et au composant attaqué. On parle de modèle « probabiliste », car il lie les paramètres de l'équipement avec les modifications observées sur le code en associant une probabilité d'occurrence à chacune des fautes

ainsi décrites.

3.5.1. Résultats

Dans cette section, on présente les inférences de modèles effectuées sur deux autres cartes A et B, avec divers équipements. Chronologiquement, on a commencé par les expériences sur carte B, puis les expériences sur carte A, et les expériences sur carte C sont venues en dernier.

3.5.1.1. Carte A

Les expériences effectuées sur la carte A sont détaillées dans [Dur+15], et sont reprises en partie dans l'annexe B.

La carte A est une carte sécurisée[9] qui possède un cœur Cortex-M3 (ARMv7-M). Elle est cadencée par une horloge externe à 4.186 MHz. Elle propose plusieurs contre-mesures matérielles (détecteur de *glitches*, détecteurs de laser). Pour la carte A, on a réutilisé l'injecteur électromagnétique utilisé pour la carte C, ainsi que le programme de détection sur la mémoire non volatile.

On obtient les ensembles de fautes suivants :

$$\{p_k \underset{f_0}{\rightsquigarrow} (i = k,\, \mathrm{read}_m(\mathrm{EEPROM}[x_k], a, 0)),\, k \in \mathbb{N} \wedge a \neq 0\}$$

$$\{p_k \underset{f_{1\%}}{\rightsquigarrow} (i = k,\, \mathrm{read}_m(\mathrm{EEPROM}[x_k], a, b)),\, k \in \mathbb{N} \wedge a \neq 0 \wedge d(a,b) \leqslant 1\%\}$$

$$\{p_k \underset{f_{20\%}}{\rightsquigarrow} (i = k,\, \mathrm{read}_m(\mathrm{EEPROM}[x_k], a, b)),\, k \in \mathbb{N} \wedge a \neq 0 \wedge d(a,b) \leqslant 20\%\}$$

$$\{p_k \underset{f_{\mathrm{rand}}}{\rightsquigarrow} (i = k,\, \mathrm{read}_m(\mathrm{EEPROM}[x_k], a, b)),\, k \in \mathbb{N} \wedge a \neq 0 \wedge d(a,b) > 20\%\}$$

$$\{p_k \underset{f_{\mathrm{double}}}{\rightsquigarrow} (i = (k, k+1),\, (\mathrm{read}_m(\mathrm{EEPROM}[x_k], a_0, 0), \mathrm{read}_m(\mathrm{EEPROM}[x_k + 1], a_1, 0))),\, k \in \mathbb{N} \wedge (a_0, a_1) \neq 0\}$$

où $d(a,b)$ est la distance relative[10] entre a et b. On en extrait alors le modèle de faute présenté au tableau 3.5.

3.5.1.2. Carte B

Dispositif expérimental. La carte B est une carte sécurisée qui utilise un processeur de type CISC. Elle est cadencée par une horloge à 29 MHz. Pour la carte B, on a réutilisé le laser utilisé pour la carte C. On a utilisé le programme de détection de faute sur la mémoire non volatile.

9. C'est-à-dire qu'elle a reçu un certificat EAL4+ dans le cadre d'une évaluation CC.

10. Il s'agit de la distance dite « en valeur » définie par $d(a,b) = \frac{|a-b|}{a}$, et non de la distance « de Hamming » qui calcule la différence des poids de Hamming de a et b. La distance de Hamming a été considérée, mais n'a pas donné de résultats.

Faute	Probabilité
f_0	4.8%
$f_{1\%}$	1.8%
$f_{20\%}$	1.6%
f_{rand}	1.3%
f_{double}	0.5%

TABLE 3.5. – Modèle de faute probabiliste pour la carte A

Faute	Probabilité
f_1	4.32%
f_2	2.93%
f_3	3.13%
f_4	2.98%
f_5	6.56%
f_6	2.48%

TABLE 3.6. – Modèle de faute probabiliste pour la carte B

Modèle inféré. Les attaques sur la carte B injectent la valeur `00` de façon beaucoup plus déterministe que pour la carte A. En revanche, le nombre de valeurs affectées est très variable : entre 1 et 6 octets consécutifs. L'espace des paramètres d'équipement considéré est :

$$\mathcal{P} = \{(I = 1\,\text{W}, s = 20\,\mu\text{m}^2, x = x_0, y = y_0, w = 100\,\text{ns}, d = t_0 + j\delta) \mid j \in \mathbb{N}\},$$

avec x_0, y_0, t_0 et δ fixés. On note f_k la faute qui perturbe k lectures EEPROM consécutives à 0. On obtient alors le modèle de faute de la table 3.6.

77.58% des attaques conduisent à un mutisme ou un cas nominal.

3.5.1.3. Variabilités des modèles de faute probabilistes

On a inféré quatre modèles de fautes probabilistes pendant la thèse, dont deux au laser, deux à l'injecteur électromagnétique, et deux sur la même carte.

On remarque plusieurs points communs et différences entre ces modèles :

— Importance de la valeur `00` : dans tous les modèles de faute, sur les composants testés, la valeur `00` joue un rôle particulier. Il existe en effet toujours des paramètres permettant de l'injecter. Par ailleurs, il est plus difficile (mais pas impossible sur la carte C) de perturber la valeur `00` en une autre valeur.

— Importance de la valeur `FF` : dans les modèles de faute de la carte C, la valeur `FF` semble jouer un rôle particulier. Elle devient injectable avec les paramètres d'équipement appropriés au laser, et parfois à l'EM (suivant des paramètres difficiles à contrôler).

— Importance des autres valeurs : seule la carte A permet d'injecter régulièrement des valeurs différentes de `00` et `FF`.

— Largeur des attaques : ce facteur est un des plus variables entre les modèles de faute. On obtient des modèles qui écrasent presque toujours 1 demi-mot, des modèles qui écrasent entre 1 et 5 octets, et des modèles qui écrasent systématiquement 16 octets.
— Impact de l'équipement : pour évaluer ce facteur, on ne dispose que des expériences menées sur carte C à la fois à l'EM et au laser. On observe une moins grande variabilité des valeurs perturbées au laser, et des paramètres spatiaux mieux délimités.

Sur ces échantillons de cartes, la variabilité des modèles de fautes inférés nous paraît suffisante pour justifier de spécialiser les modèles de fautes observés en fonction du composant attaqué. L'influence de l'équipement pourrait en revanche être moins prégnante qu'initialement estimé.

3.5.2. Bilan de la méthode proposée

Les démarches visant à établir des modèles de faute à partir d'attaques par perturbation répondent à un problème précis, celui de les lier avec la simulation des fautes résultantes sur le code. Un point crucial de différenciation par rapport aux méthodes existantes est que l'inférence de modèles de faute a été formalisée en tant que cadre applicable de façon générique aux composants sécurisés. En particulier, on a pu constater que les modèles dépendent du composant attaqué, voire de l'équipement utilisé. Autre différence, la méthode a été appliquée à plusieurs composants de niveaux de sécurité différents (cartes A et C) et d'architectures différentes (ARMv7-M et de type CISC). Les résultats semblent valider le lien entre équipement, composant et modèle. En revanche, la méthode ne s'affranchit pas de l'explosion combinatoire des paramètres d'équipement et du code, que toutes les méthodes doivent gérer. Ainsi, l'extraction de modèles de faute a recours au même type de réduction de l'espace des paramètres d'équipement pour fonctionner, et Moro *et al.* doivent renoncer à la simulation exhaustive des remplacements d'instructions. Pour ce problème d'échantillonnage, il serait intéressant d'étudier l'impact de l'utilisation de méthodes probabilistes telles que la méthode de Monte-Carlo. Enfin, la phase itérative, où l'on réutilise les connaissances obtenues au fur et à mesure du processus, constitue un des aspects novateurs de la méthode proposée. De façon générale, on a essayé de découpler au maximum les conséquences directes de la perturbation sur le composant (faute) des conséquences « fonctionnelles », qui dépendent nécessairement de l'application considérée.

Objectif atteint ?

On a présenté les défis à relever par notre approche dans la section 3.3.1.3. On regarde maintenant à quel point notre approche y répond.

1. Le problème de la taille de l'espace des paramètres. Réduire la taille de l'espace des paramètres nécessite toujours l'expertise d'un évaluateur. Les phases de généralisation de modèle permettent de récupérer un peu de ce qu'on a perdu pendant la réduction, mais au prix d'approximations dont la justesse est difficile à mesurer (puisque leur but est d'éviter les mesures) et qui pourraient s'avérer gênantes. De façon générale, beaucoup

d'éléments sont manuels dans la méthode, et nécessitent un haut niveau d'expertise. Il est sans doute possible d'automatiser en partie la phase itérative, en laissant la machine générer les nouvelles hypothèses ainsi que les programmes utilisés pour les tester. De la même façon, les connaissances préalables des experts pourraient être formalisées et utilisées pour guider une phase d'apprentissage des paramètres d'équipement intéressants.

2. L'influence du composant et de l'équipement. Pour faire face à l'influence du composant ou à celle de l'équipement, l'approche d'inférence requiert de refaire les expériences (redérouler la méthode). Sur la carte C, on a pu constater que la différence de modèle entre laser et électromagnétique semble tenir à un meilleur contrôle des fautes obtenues et à une plus forte probabilité de succès au laser. Ce résultat a besoin d'être confirmé par d'autres expériences (notamment sur des cartes avec des contre-mesures matérielles), mais s'il se confirme, il pointe vers une réutilisabilité des modèles obtenus avec différents équipements.

3. L'influence du programme considéré. En proposant la notion de programmes de détection de faute (et plusieurs exemples utiles de tels programmes), on fait un premier pas dans l'indépendance de la méthode par rapport au programme considéré. On adopte aussi un formalisme de fautes qui expriment des paramètres c en fonction de p. Cependant la démarcation entre ces relations générales et des spécificités dues aux programmes de détection de faute exécutés n'est pas toujours très claire. Ainsi, on peut certes associer un délai d'attaque à un numéro de lecture EEPROM durant l'exécution, mais tous les programmes n'espacent pas nécessairement de la même manière leurs lectures, ce qui peut introduire des modifications dans la relation entre le délai et le numéro de lecture attaqué dans d'autres programmes. Dans la pratique, la méthode se contente donc de prouver l'existence de paramètres d'équipement p tels que la faute c est réalisable, mais ne donne pas toujours explicitement les valeurs de p. Là encore, réutiliser l'expertise des évaluateurs, à savoir la détermination de points de synchronisation communs sur des exécutions entre les deux espaces de paramètres, permettrait probablement d'améliorer la précision et d'expliciter des intervalles pour p, améliorant ainsi l'indépendance au programme considéré.

3.5.3. Utilisation du modèle de faute probabiliste

L'inférence de modèle de faute produit un modèle de faute probabiliste, qui est spécifique du composant et de l'équipement d'attaque. Les outils d'analyse de vulnérabilités à l'injection de fautes présentés au chapitre 2 utilisent un ensemble prédéfini de modèles de fautes « génériques », c'est-à-dire indépendamment des conditions de l'expérience. Comment utiliser un modèle de faute probabiliste ? Le chapitre 4 présente CELTIC, un outil paramétrable, qui permet d'une part d'adapter les fautes réalisées à celles qui sont possibles d'après le modèle de faute choisi, et d'autre part de calculer des métriques de robustesse quantitatives à partir des probabilités décrites dans le modèle.

Chapitre 4

CELTIC : un outil pour l'évaluateur

Savez-vous seulement quelle différence il y a entre un psychotique et un névrosé ? Un psychotique, c'est quelqu'un qui croit dur comme fer que 2 et 2 font 5, et qui en est pleinement satisfait. Un névrosé, c'est quelqu'un qui sait pertinemment que 2 et 2 font 4, et ça le rend malade.

(Pierre Desproges, Textes de scène)

Résumé du chapitre

Dans ce chapitre on présente CELTIC, l'outil d'analyse de vulnérabilités conçu et implémenté pendant la thèse. CELTIC a servi de fil rouge guidant les réflexions sur les diverses problématiques de recherche, et également de banc de test pour valider l'efficacité des solutions proposées. C'est aussi un outil conçu pour répondre aux problèmes d'injection de fautes complexes et concrets rencontrés par les évaluateurs CESTI. Ce chapitre commence donc par établir précisément les besoins des évaluateurs, dessiner les contours des réponses de l'outil à ces besoins, détailler le fonctionnement de l'outil et décrire un exemple d'utilisation. Un premier besoin majeur identifié est la polyvalence de l'outil : les évaluateurs doivent s'adapter aux composants et scénarios variés des développeurs. Pour y répondre, on a doté CELTIC de nombreux points d'entrées paramétrables, l'un des plus importants étant la possibilité d'ajouter des modèles de fautes. Un autre besoin majeur est un traitement efficace des résultats afin que l'évaluateur puisse émettre un diagnostic le plus fiable possible sur la sécurité à partir de l'analyse. CELTIC réutilise les modèles de fautes probabilistes introduits au chapitre 3 pour calculer des métriques quantitatives de la sécurité d'une implémentation. Un autre point de paramétrage important de CELTIC est le langage de spécification GISL, qui permet d'ajouter de nouveaux composants. Enfin, on rapporte quelques données d'utilisation de l'outil au CESTI, et on ouvre sur le chapitre suivant qui s'occupe du cas des fautes multiples indépendantes.

Sommaire

4.1. Introduction

Au chapitre 2, on a vu que le problème de l'injection de fautes a promu le développement de nombreux outils, tant pour l'analyse de tolérance aux fautes que pour la détection de vulnérabilité aux attaques par perturbation.

Cependant l'analyse de vulnérabilités aux attaques par perturbation s'inscrit notamment dans le cadre plus large de la certification des composants sécurisés (comme présenté au chapitre 1), et fait en particulier partie des tâches dévolues aux CESTI durant leurs évaluations. L'analyse de vulnérabilités à l'injection de fautes pour la certification représente un besoin particulier qui découle à la fois des attendus de l'évaluation et du contexte des laboratoires CESTI.

4.1.1. Besoin d'un outil pour l'évaluation

Pour rappel, on a vu à la section 1.3.3 que le processus d'évaluation peut s'articuler suivant une phase de revue de code et une phase de tests de pénétration.

4.1.1.1. Motivation

Les outils d'analyse de code pour la recherche de vulnérabilités interviennent naturellement en complément de, ou en alternative à, la revue de code manuelle. En effet, la revue de code repose traditionnellement sur l'expertise des évaluateurs CESTI. Cette dépendance freine la reproductibilité des évaluations. Or, les organismes délivrant officiellement les certificats requièrent et vérifient la capacité des différents laboratoires CESTI à fournir le même résultat lors des évaluations. Il est donc important de développer des méthodes permettant d'assurer la reproductibilité des analyses. À ce titre, un outil automatique déplace l'expertise depuis la vérification systématique vers la configuration de l'outil et l'analyse de ses résultats.

Par ailleurs, certaines classes de vulnérabilités sont particulièrement difficiles à détecter manuellement. C'est le cas par exemple des vulnérabilités introduites par les fautes qui dépendent de l'encodage binaire du programme. Le tableau 4.1 donne un exemple d'une telle faute. La ligne L1 du tableau présente une suite d'instructions avec les octets correspondants dans le binaire[1]. Ainsi, `0x12` correspond au code de l'opération LD_X, qui charge un octet dans le registre X. La valeur de cet octet est spécifiée dans l'opérande de cette instruction. La suite d'octets `0x12 0x2a` correspond ainsi à l'instruction qui charge la valeur `0x2a` dans le registre X. La ligne L2 du tableau présente le même code binaire, à l'exception du troisième octet, qui a été remplacé par la valeur `0x82` (au lieu de `0x14`). L'instruction à une opérande qui charge la valeur `0x32` dans le registre A est ainsi remplacée par l'instruction à deux opérandes qui effectue un saut conditionnel vers l'adresse `0x3223`. Ce remplacement d'une instruction à une opérande par une instruction à deux opérandes a pour conséquence que l'octet `0x23`, précédemment interprété comme le code d'opération ADD_A, est interprété comme le second opérande de l'instruction précédente. Ceci décale l'interprétation des opérandes et des instructions, et fait surgir de nouvelles instructions « fantômes » dans la suite du

1. Les codes opérations choisis et les modes d'adressages sont fantaisistes.

L1	0x12 0x2a LD_X 2a	0x14 0x32 LD_A 32	0x23 0x12 ADD_A 12	0x33 ADD_A X
L2	0x12 0x2a LD_X 2a	**0x82** 0x32 0x23 JNZ 3223		0x12 0x33 LD_X 33

TABLE 4.1. – Exemple de faute dépendante de l'encodage

code (ici, l'instruction LD_X, 33). Ces instructions causées par le décalage d'interprétation peuvent avoir des effets très divers sur le code, et bien entendu ont le potentiel de compromettre les propriétés de sécurité. Malheureusement, les instructions fantômes constituent un phénomène très difficile à prendre en compte pour un évaluateur, qui travaille typiquement aux niveaux source et assembleur, où les instructions fantômes ne sont pas visibles.

Une troisième motivation est la possibilité (discutée dans la section 1.2) pour les attaquants d'effectuer plusieurs attaques par perturbation au cours d'une même exécution, ce qui résulte en l'introduction de fautes multiples indépendantes [2]. Pour tenir compte de toutes les fautes multiples exhaustivement, il faut donc vérifier toutes les combinaisons de fautes possibles, ce qui devient rapidement irréalisable dans le cadre d'une évaluation manuelle.

4.1.1.2. Contraintes

L'environnement CESTI impose des contraintes particulières sur l'outil d'analyse de code. Dans le cadre de la thèse, on a relevé les contraintes suivantes :

Non-intrusion. Il est important pour le CESTI de pouvoir effectuer l'analyse directement sur le code qui a été remis par les clients, sans modifications. Les approches d'instrumentation ou de recompilation du code sont donc à proscrire. De plus, il n'est pas toujours possible pour le CESTI de charger du code dans les composants évalués, la solution envisagée ne doit donc pas nécessiter d'être embarquée sur le composant.

Support de multiples composants. Le CESTI doit pouvoir s'adapter aux différents composants sécurisés soumis pour certification par les clients. L'outil proposé doit donc pouvoir fonctionner sur ces différents composants, l'idéal étant qu'il soit assez flexible pour permettre l'ajout du support de nouveaux composants par les évaluateurs CESTI eux-mêmes.

Représentativité de l'analyse. L'analyse de vulnérabilité vise à fournir un diagnostic duquel dépend en partie la délivrance du certificat du produit, et doit donc assurer un niveau de fiabilité adapté à ces enjeux (voir 1.1.3). L'analyse doit donc s'effectuer au niveau binaire, pour tenir compte de l'influence du compilateur et des fautes qui dépendent de l'encodage binaire. De plus, l'outil doit être capable d'utiliser un modèle

2. Pour rappel, on parle de fautes multiples *indépendantes* pour insister sur le fait que les fautes sont causées par plusieurs perturbations distinctes.

de faute représentatif des fautes réalisables avec l'équipement d'attaque choisi sur le composant considéré, c'est-à-dire un modèle de faute obtenu avec une approche similaire à celle présentée au chapitre 2.

Traitement des résultats. Dans l'optique d'émettre le diagnostic le plus précis possible sur la robustesse de l'application, la méthode de traitement des résultats de l'analyse de robustesse est primordiale. Comme indiqué dans la section 2.2.5, l'utilisation de modèles de faute tend à produire beaucoup de résultats qu'il faut donc classer pour pouvoir conclure sur la robustesse du produit évalué. L'outil doit être capable de déterminer automatiquement qu'une faute conduit à une attaque réussie, voire idéalement de classer les vulnérabilités découvertes en fonction de la menace qu'elles représentent pour l'application. D'autre part, le CESTI doit associer une cotation à chaque attaque découverte. Cette cotation, qui est la somme de plusieurs sous-facteurs, est traditionnellement calculée à l'aide des résultats des tests de pénétration, sans utiliser la revue de code. La détermination de métriques de robustesse associées à un logiciel embarqué peut également aider à la cotation des attaques découvertes. Il est donc important qu'un outil conçu pour l'évaluation implémente de telles métriques.

Passage à l'échelle de l'outil. L'outil d'aide à l'évaluation doit être dimensionné pour fonctionner dans un contexte d'évaluation réel, c'est-à-dire qu'il doit être capable d'analyser les fonctions de sécurité de l'application en un temps raisonnable. Idéalement, ce temps doit être plus court que l'analyse manuelle, mais il est acceptable qu'il soit plus long car l'outil peut fonctionner hors des heures travaillées. La base de code à traiter n'est pas aussi importante que dans d'autres applications, car les codes embarqués doivent rester courts et s'inscrivent dans un écosystème fermé, et que seules les fonctions assurant des propriétés de sécurité sont analysées. En revanche, le problème de l'injection de faute complexifie l'analyse, puisqu'il est nécessaire de tenir compte des modifications d'exécution causées par chaque faute. Ce problème est, comme pour l'analyse manuelle, exacerbé en présence de fautes multiples.

4.1.2. Présentation générale de l'outil

CELTIC [3] est la réponse du CESTI-LETI au besoin énoncé dans cette introduction.

L'outil est un simulateur dynamique de code binaire, capable d'injecter des fautes durant la simulation du code. L'approche de simulation permet la non-intrusion, puisque le simulateur s'exécute sur un ordinateur classique et non directement sur carte, et qu'il travaille directement sur le binaire fourni pour l'évaluation (la modification du binaire ou la recompilation du code source ne sont pas nécessaires). Le choix du niveau binaire améliore également la représentativité de l'analyse en permettant aux modèles de fautes d'opérer à ce niveau. Par ailleurs, CELTIC supporte les modèles de faute probabilistes présentés au chapitre 2. Le fonctionnement du simulateur est détaillé à la section 4.2. La section 4.3 précise la notion de fautes telle qu'utilisée dans l'analyse, l'impact des fautes sur la simulation et leur implémentation dans CELTIC.

3. « CEsti-LeTi Integrated Circuits », un acronyme qui s'est révélé être malheureux quand j'ai découvert que CESTI se dit ITSEF en Anglais.

Le traitement des résultats utilise la notion « d'oracle », c'est-à-dire qu'après chaque exécution fautée l'état du processeur simulé est inspecté pour déterminer s'il vérifie un prédicat donné. De plus, CELTIC réutilise les modèles de fautes probabilistes pour calculer des métriques pertinentes de robustesse du code. Ces métriques sont détaillées dans la section 4.4.

Enfin, le support de plusieurs composants est assuré par un *Architecture Description Language* ou "langage de spécification d'architecture" (ADL) qui permet de spécifier les caractéristiques matérielles du composant ainsi que son jeu d'instruction. Le langage de spécification est abordé dans la section 4.5.

4.1.3. L'œil de l'évaluateur : exemple « fil rouge »

Tout au long de ce chapitre, on va se servir d'une implémentation de `VerifyPIN` comme exemple pour illustrer l'utilisation de l'outil par un évaluateur et comparer avec la revue de code manuelle. L'implémentation est tirée de la *Fault Injection and Simulation Secure Collection* (FISSC) [4] qu'on a proposée dans le cadre du projet SERTIF [5] [Dur+16]. Dans la collection, il s'agit précisément de la version 2 de `VerifyPIN`. Le listing de code 4.1 donne le code source correspondant à l'exemple. Le code assembleur correspondant à une compilation vers une carte d'architecture ARMv7-M sont donnés par les listings 4.2 et 4.3.

Dans cette implémentation, on remarque plusieurs particularités relatives à la sécurité :

— La contre-mesure de vérification de l'indice de boucle présentée au chapitre 1 est présente dans `byteArrayCompare`. La fonction `countermeasure()` signale la détection d'une attaque [6].

— Le jeton d'authentification ne prend pas une valeur non nulle quelconque pour indiquer `true` et 0 pour indiquer `false` comme c'est habituellement le cas en `C`, mais respectivement `BOOL_TRUE` (`0xAA`) et `BOOL_FALSE` (`0x55`). Ces booléens « renforcés » permettent des comparaisons plus précises (une seule valeur correcte pour `true`), et n'utilisent pas les valeurs spéciales `0` ou `0xff` qui peuvent s'injecter plus facilement.

4. Disponible sur requête envoyée à l'adresse `sertif-secure-collection@imag.fr`

5. Simulation pour l'Evaluation de la RobusTesse des applications embarquées contre l'Injection de Fautes, ANR-14-ASTR-0003-01, `http://sertif-projet.forge.imag.fr/`

6. Dans les exemples de FISSC, un comportement possible pour la fonction `countermeasure` est de simplement assigner la variable `g_countermeasure` à 1. Ce comportement est utile pour les outils car il permet de connaître l'état final de simulation même si une contre-mesure est levée. Un code destiné à une carte à puce cause plutôt son mutisme ou sa destruction. Toutefois, ce comportement n'affecte pas la robustesse attribuée au code, car celle-ci ne tient compte que du fait qu'on appelle ou non `countermeasure()`, indépendamment de sa sémantique (c'est-à-dire qu'on fait l'hypothèse que l'effet de `countermeasure()` ne peut pas être contré par l'attaquant).

Exemple 4.1 *Revue de code manuelle.*
Sans prétendre être évaluateur, on peut se prêter à l'exercice de la revue de code, pour donner une idée des résultats attendus. On se place dans un scénario d'attaque où l'attaquant ne dispose pas de la bonne valeur du code PIN, mais essaie d'obtenir de s'authentifier malgré cela. Dans le code, ce scénario correspond à avoir le jeton d'authentification `g_authenticated` à `BOOL_TRUE`, alors que les contenus de `g_userPin` et `g_cardPin` sont différents. En appliquant manuellement un modèle de faute de saut d'instruction sur l'assembleur (voir listings 4.2 et 4.3), on trouve les vulnérabilités suivantes :

— Supprimer l'instruction de retour de la fonction `byteArrayCompare` à l'adresse `0x41ce` (`pop {r4, r5, r6, pc}` [a]) permet d'exécuter en séquence l'affectation de la ligne 22 du source (`status = BOOL_FALSE`) et l'affectation de la ligne 20 (`status = BOOL_TRUE`) avant le retour de la fonction, ce qui a pour effet de mettre la valeur `BOOL_TRUE` dans la variable `status`, et donc de faire retourner `BOOL_TRUE` à la fonction `byteArrayCompare`, ce qui conduit à la victoire.

— De la même façon, supprimer l'instruction de retour de fonction d'adresse `0x41fc` (à la fin de `VerifyPIN`) revient à exécuter en séquence les deux branches du *statement* conditionnel ligne 32, en commençant par la branche `else` (par le hasard de la compilation). Ceci conduit donc à exécuter en dernier le code d'authentification, et donc à une authentification réussie.

— Supprimer l'instruction de comparaison d'adresse `0x41c8` (`cmp r4, #85`) réutilise l'état du registre de status obtenu après la comparaison précédente d'adresse `0x41bc` (`cmp r3, r2`), qui avait permis de faire le saut conditionnel d'adresse `0x41c2` (`beq.n 41c8`). Ceci permet de faire le saut conditionnel d'adresse `0x41ca` (`beq.n 41d0`), ce qui conduit à la victoire. Sur le source, ceci est équivalent à forcer la condition `diff == BOOL_FALSE` ligne 19 à `true`, pour exécuter l'affectation ligne 20 au lieu de celle ligne 22.

Le processus ne dit rien sur la dangerosité effective de ces trois vulnérabilités, mais on constate l'absence de contre-mesures permettant d'empêcher ces sauts. On peut donc considérer les vulnérabilités comme bien présentes. On gardera ces résultats à l'esprit dans la suite du chapitre.

a. En ARM, il est courant d'utiliser une instruction `PUSH {..., lr}` en début de fonction pour enregistrer l'adresse de retour, et une instruction `POP {..., pc}` en fin de fonction pour récupérer l'adresse de retour dans le registre `PC`.

</>Code source

```
1  extern SBYTE g_ptc;
2  extern BOOL g_authenticated;
3  extern UBYTE g_userPin[PIN_SIZE];
4  extern UBYTE g_cardPin[PIN_SIZE];
5
6  BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size)
7  {
8    int i;
9    BOOL status = BOOL_FALSE;
10   BOOL diff = BOOL_FALSE;
11   for (i = 0; i < size; i++) {
12     if (a1[i] != a2[i]) {
13       diff = BOOL_TRUE;
14     }
15   }
16   if (i != size) {
17     countermeasure();
18   }
19   if (diff == BOOL_FALSE) {
20     status = BOOL_TRUE;
21   } else {
22     status = BOOL_FALSE;
23   }
24   return status;
25 }
26
27 BOOL verifyPIN_2()
28 {
29   g_authenticated = BOOL_FALSE;
30
31   if (g_ptc > 0) {
32     if (byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE) == BOOL_TRUE) {
33       g_ptc = 3;
34       g_authenticated = BOOL_TRUE; // Authentication();
35       return BOOL_TRUE;
36     } else {
37       g_ptc--;
38       return BOOL_FALSE;
39     }
40   }
41   return BOOL_FALSE;
42 }
```

Listing 4.1. – Code source de l'implémentation testée

</>Code assembleur

```
041a8 <byteArrayCompare>:
 41a8:        b570             push        {r4, r5, r6, lr}
 41aa:        2455             movs        r4, #85        ; 0x55
 41ac:        2300             movs        r3, #0
 41ae:        e005             b.n        41bc <byteArrayCompare+0x14>
 41b0:        5cc5             ldrb        r5, [r0, r3]
 41b2:        5cce             ldrb        r6, [r1, r3]
 41b4:        42b5             cmp        r5, r6
 41b6:        d000             beq.n        41ba <byteArrayCompare+0x12>
 41b8:        24aa             movs        r4, #170        ; 0xaa
 41ba:        1c5b             adds        r3, r3, #1
 41bc:        4293             cmp        r3, r2
 41be:        dbf7             blt.n        41b0 <byteArrayCompare+0x8>
 41c0:        4293             cmp        r3, r2
 41c2:        d001             beq.n        41c8 <byteArrayCompare+0x20>
 41c4:        f000 f828        bl        4218 <countermeasure>
 41c8:        2c55             cmp        r4, #85        ; 0x55
 41ca:        d001             beq.n        41d0 <byteArrayCompare+0x28>
 41cc:        2055             movs        r0, #85        ; 0x55
 41ce:        bd70             pop        {r4, r5, r6, pc}
 41d0:        20aa             movs        r0, #170        ; 0xaa
 41d2:        bd70             pop        {r4, r5, r6, pc}
```

Listing 4.2. – Code assembleur de la fonction `byteArrayCompare`

</>Code assembleur

```
041d4 <verifyPIN_2>:
 41d4:        b570             push        {r4, r5, r6, lr}
 41d6:        4d0c             ldr        r5, [pc, #48]        ; (4208 <verifyPIN_2+0x34>)
 41d8:        4c0c             ldr        r4, [pc, #48]        ; (420c <verifyPIN_2+0x38>)
 41da:        2055             movs        r0, #85        ; 0x55
 41dc:        7028             strb        r0, [r5, #0]
 41de:        f994 0000        ldrsb.w        r0, [r4]
 41e2:        2800             cmp        r0, #0
 41e4:        dd09             ble.n        41fa <verifyPIN_2+0x26>
 41e6:        2204             movs        r2, #4
 41e8:        4909             ldr        r1, [pc, #36]        ; (4210 <verifyPIN_2+0x3c>)
 41ea:        480a             ldr        r0, [pc, #40]        ; (4214 <verifyPIN_2+0x40>)
 41ec:        f7ff ffdc        bl        41a8 <byteArrayCompare>
 41f0:        28aa             cmp        r0, #170        ; 0xaa
 41f2:        d004             beq.n        41fe <verifyPIN_2+0x2a>
 41f4:        7820             ldrb        r0, [r4, #0]
 41f6:        1e40             subs        r0, r0, #1
 41f8:        7020             strb        r0, [r4, #0]
 41fa:        2055             movs        r0, #85        ; 0x55
 41fc:        bd70             pop        {r4, r5, r6, pc}
 41fe:        2003             movs        r0, #3
 4200:        7020             strb        r0, [r4, #0]
 4202:        20aa             movs        r0, #170        ; 0xaa
 4204:        7028             strb        r0, [r5, #0]
 4206:        bd70             pop        {r4, r5, r6, pc}
 4208:        10008014         .word        0x10008014
 420c:        10008015         .word        0x10008015
 4210:        1000801b         .word        0x1000801b
 4214:        10008017         .word        0x10008017
```

Listing 4.3. – Code assembleur de la fonction `VerifyPIN`

4.2. Simulation

Dans cette section, on décrit les algorithmes de simulation et d'injection de fautes implémentés dans CELTIC.

4.2.1. Analyse dynamique

Le choix d'un algorithme dynamique plutôt que statique s'est imposé pour plusieurs raisons : l'analyse statique de binaires est un problème ouvert, indépendamment de l'injection de fautes, surtout si l'on vise la paramétrisation par le jeu d'instruction.

4.2.1.1. Choix de conception

L'implémentation de la simulation a conduit à favoriser certains critères au détriment des autres. La figure 4.1 illustre les différents critères dans la balance. Le **contrôle de la simulation**, qui a été nettement favorisé dans la conception de CELTIC, permet d'obtenir un affichage détaillé des opérations effectuées par le simulateur, et rend également possible l'injection de fautes. Favoriser le contrôle a néanmoins un impact direct sur la **rapidité**, qui reste un facteur important dans CELTIC, car on souhaite que la simulation puisse traiter les programmes évalués au CESTI. Offrir un excellent contrôle tout en gardant une rapidité correcte nous a donc conduit à limiter la **précision** de l'analyse : ainsi, les interruptions, les coprocesseurs, les entrées sorties, la MMU, etc., ne sont pas simulés. De plus, la simulation est précise à l'instruction près, et non au cycle près[7]. La précision retenue est néanmoins adaptée au problème, car l'évaluation s'intéresse uniquement aux fonctions de sécurité du composant, donc à un jeu restreint de fonctions.

4.2.1.2. Algorithme de simulation

L'algorithme de simulation développé dans CELTIC est une boucle classique de simulation, qui reprend les algorithmes dynamiques proposés depuis au moins les années 80 [SW79 ; Cat86 ; May87].

La figure 4.2 illustre cette boucle de simulation, tandis que le listing 4.4 en donne une version en pseudo-code (de style `python`). La boucle est constituée de deux opérations `decode()` et `exec()`, qui agissent tour à tour sur l'état de la simulation. L'état de la simulation est composé de la valeur instantanée des registres et de la mémoire de la machine simulée, et contient en particulier le registre ordinal (dit registre `PC`, pour *Program Counter*) qui indique l'adresse dans le code de la prochaine instruction à exécuter. La fonction `decode()` interprète les octets du binaire à partir de l'adresse `PC` comme du code afin de déterminer l'instruction suivante à exécuter. Une fois l'instruction décodée, la fonction `exec()` met à jour l'état de la simulation suivant la sémantique de l'instruction. La sémantique retenue est à granularité fine : chaque instruction se traduit par une suite d'opérations atomiques sur l'état consistant en des calculs arithmétiques et des lectures/écritures dans des emplacements mémoires. L'initialisation de la boucle se fait à partir d'un état initial s_0 fourni par l'utilisateur (comprenant une valeur initiale

7. Ceci facilite également le support de nouveaux composants pour le simulateur.

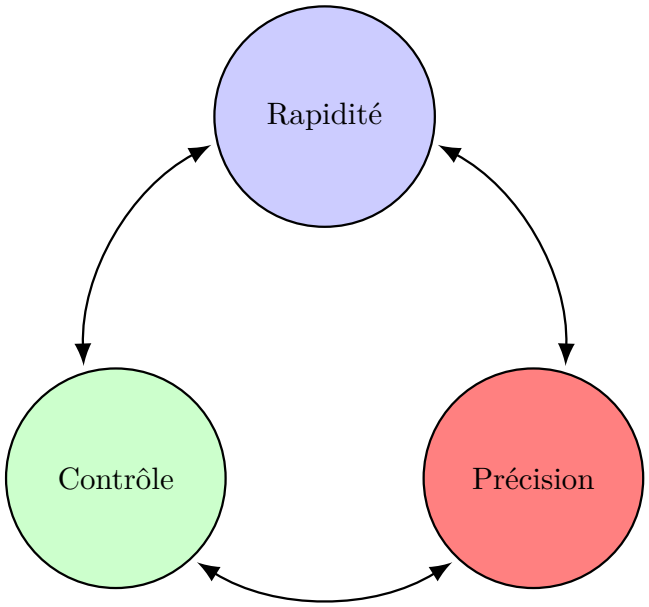


FIGURE 4.1. – Choix de conception du simulateur

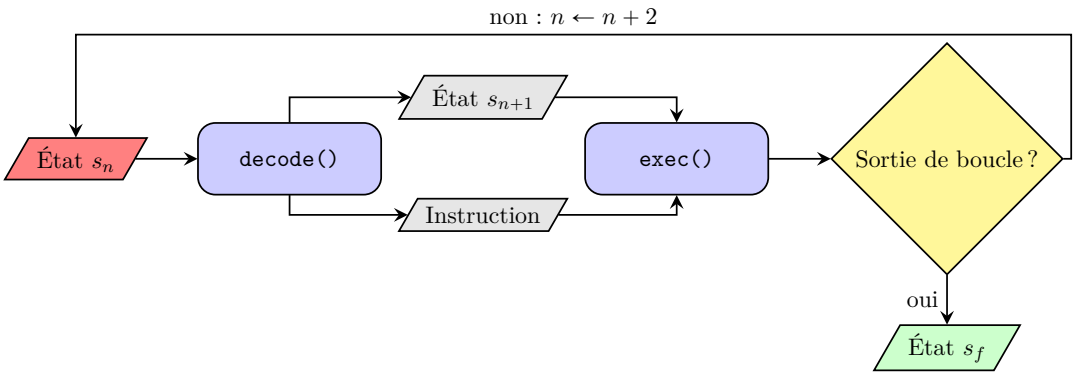


FIGURE 4.2. – Algorithme de simulation

pour PC). La boucle de simulation se termine dès que l'une des conditions suivantes est atteinte :

— après un nombre d'itérations prédéfini par l'utilisateur (`iter_max`) ;
— si une erreur est détectée dans l'état (lecture d'un emplacement mémoire non initialisé, division par zéro, etc.) ou
— si le registre PC de l'état atteint une valeur prédéfinie par l'utilisateur (`end_addr`).

</>Code source

```
def simulate(simu, initial_state, iter_max, end_addr):
    iter = 0
    state = initial_state.copy
    trace = []
    while (state.pc != end_addr and
        not state.hasError and
        iter < iter_max):
            iter = iter + 1
            inst = simu.fetch(state, trace)
            simu.exec(state, inst, trace)
    return trace, state
```

Listing 4.4. – Algorithme de simulation

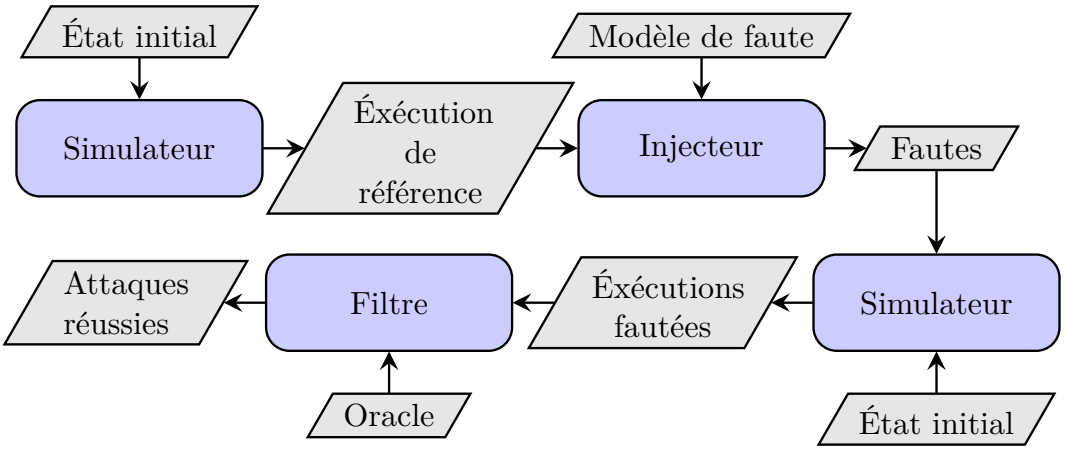


FIGURE 4.3. – Algorithme d'injection de fautes

Exemple 4.2
Sur notre exemple fil rouge, on lance une simulation avec un état initial contenant le code listé en 4.2 et 4.3, en initialisant les registres à 0, sauf `PC` à `0x41d4` (adresse de début du listing 4.3) et en choisissant l'adresse de retour de `verifyPIN` comme adresse de fin de simulation. On obtient le comportement suivant :

1. Au début de la simulation, la fonction `decode()` est appelée, elle renvoie l'instruction `push {r4, r5, r6, lr}` et incrémente `PC` jusqu'à `0x41d6`.
2. L'instruction `push {r4, r5, r6, lr}` est exécutée, ce qui a pour effet de sauvegarder le contenu des registres `r4`, `r5`, `r6` et `lr` dans la pile.
3. La fonction `decode()` est à nouveau appelée et elle renvoie cette fois l'instruction `ldr r5, [pc, #48]` en incrémentant `PC` à `0x41d8`.
4. L'instruction `ldr, r5, [pc, #48]` est exécutée, ce qui a pour effet de charger l'adresse de la variable `g_authenticated` dans `r5`.
5. L'exécution se poursuit de la même manière pour les instructions suivantes.
6. Lorsque le registre `PC` atteint la valeur de l'adresse de retour de `VerifyPIN` (après l'exécution de l'un des `pop` aux adresses `0x41fc` ou `0x4206`), l'exécution se termine.

À l'issue de l'exécution, la liste des instructions exécutées ainsi que la liste des opérations élémentaires effectuées par le simulateur sont disponibles pour l'utilisateur. De plus, l'évaluateur peut faire des requêtes de lecture sur l'état résultant de la simulation. Par exemple, il peut ainsi vérifier qu'en l'absence d'attaque, la variable `g_authenticated` vaut bien `BOOL_FALSE` à l'issue de l'appel à `VerifyPIN`.

4.2.1.3. Algorithme d'injection de faute

L'algorithme d'injection de faute utilisé est également classique par rapport aux autres outils d'analyse dynamique pour l'injection de faute [And09 ; Ber+14 ; Höl+15].

Diagnostic	Description
Erreur	L'exécution se termine en erreur.
Attaque nominale	L'exécution se termine sans erreur, mais l'oracle n'est pas vérifié par l'état final de simulation.
Attaque réussie	L'exécution se termine sans erreur, et l'oracle est vérifié par l'état final de simulation.
Contre-mesure	L'exécution se termine sans erreur, mais une contre-mesure a détecté l'attaque.

TABLE 4.2. – Diagnostics possibles en fin de simulation

Il repose sur l'algorithme de simulation proposé à la section précédente. La figure 4.3 illustre les différentes étapes de l'injection de faute, tandis que le listing 4.5 donne le pseudo-code correspondant à l'algorithme. Une première simulation est effectuée sans injection de faute, on parle alors de *golden run*. Le *golden run* produit une trace d'exécution de référence (la liste des opérations effectuées durant la simulation, dont les instructions exécutées). L'injecteur parcourt l'exécution de référence pour générer toutes les fautes injectables, conformément au modèle de faute. Enfin, pour chaque faute, une simulation est effectuée, où les opérations ciblées par la faute sont modifiées suivant la faute injectée. On utilise un oracle (c'est-à-dire un prédicat sur l'état de la simulation) pour classer les exécutions fautées en attaque réussie ou en échec. Plus précisément, pour chaque attaque, on détermine l'un des quatre diagnostics suivants : erreur, attaque nominale, attaque réussie, ou contre-mesure. Les diagnostics sont détaillés dans la table 4.2. Dans la pratique, la détection de la contre-mesure est souvent intégrée à l'oracle quand c'est possible (par exemple sur l'exemple fil rouge, on peut intégrer la condition `g_countermeasure != 1` à l'oracle). On revient sur la notion d'oracle à la section 4.2.3.

</>Code source

```
def inject(simu, initial_state, iter_max, end_addr, fault_model, oracle):
    attacks = []
    successful_attacks = []
    golden_run, golden_state = simulate(simu, initial_state,
            iter_max, end_addr)
    attacks = fault_model.generate(golden_run)
    for attack in attacks:
        state = initial_state.copy
        simu.attach(state, attack)
        attack_run, attacked_state = simulate(simu, state,
                iter_max, end_addr)
        if (simu.filter(oracle, attack_run, attacked_state)):
            successful_attacks.push(attack)
    return attacks, successful_attacks
```

Listing 4.5. – Algorithme d'injection de fautes

Exemple 4.3
Sur notre exemple fil rouge on lance une campagne d'injection de fautes. On choisit le modèle de fautes NOP, qui remplace une instruction quelconque par une instruction NOP. Notre oracle est le prédicat :

```
g_authenticated == BOOL_TRUE && g_countermeasure == 0
```

qui exprime que la variable d'authentification est à la valeur `BOOL_TRUE` et que la variable qui indique si une contre-mesure a été rencontrée est `false`. On effectue 62 exécutions fautées (une par instruction exécutée). À l'issue de chaque simulation fautée, CELTIC vérifie l'état de `g_authenticated` et `g_countermeasure`, et si le prédicat est vérifié, l'attaque est considérée réussie. Pour chaque attaque réussie, CELTIC fournit à l'utilisateur les caractéristiques de la faute associée (ses paramètres dans l'espace du code), ainsi que la trace d'exécution fautée. Ceci permet, en la comparant à la trace d'exécution de référence, de comprendre pourquoi la faute cause la réussite de l'attaque.

Avec le modèle NOP, on obtient 3 attaques réussies sur les 62 effectuées : on retrouve les trois attaques identifiées aux adresses `0x41c8`, `0x41ce` et `0x41fc` pendant la revue manuelle.

Sur les modèles de faute simples, CELTIC permet de confirmer les résultats des revues de codes manuelles.

4.2.2. Le problème de l'initialisation

L'algorithme de simulation qu'on vient de présenter nécessite en entrée un état initial s_0 fourni par l'utilisateur. Cet état initial est construit en initialisant les registres et la mémoire à des valeurs compatibles avec le contexte d'appel de la fonction. Ainsi, les variables globales et les paramètres de la fonction, qui peuvent être issus de nombreux calculs précédents, doivent être initialisés par l'utilisateur avec des valeurs représentatives de celles qu'elles auraient au début de l'appel de la fonction simulée.

Exemple 4.4
Dans notre exemple fil rouge, il faut initialiser `g_userPin` et `g_cardPin`, qui correspondent respectivement au code PIN saisi par l'utilisateur et au code PIN de référence dans la mémoire de la carte, à des valeurs différentes, pour jouer un scénario d'attaque dans lequel l'attaquant ne connaît pas le code PIN.

Ce problème d'initialisation existe en fait déjà lors de la revue manuelle, où l'évaluateur fait le même type d'hypothèses sur les valeurs contenues dans les variables, mais il est rendu plus complexe dans CELTIC, car l'outil ne manipule que des données concrètes instanciées. Lors de la revue manuelle, l'évaluateur peut avoir recours à des initialisations « floues », c'est-à-dire supposer que les variables ont des valeurs qui permettent au code de fonctionner, sans préciser ces valeurs. Dans CELTIC, les valeurs d'initialisation doivent être données explicitement pour permettre l'exécution.

Ce problème pourrait être atténué en simulant le composant depuis son démarrage et jusqu'à l'appel de la fonction. Dans ce cadre, on bénéficierait alors du contexte d'appel

« naturel » de la fonction. Néanmoins cette approche n'est en général pas praticable pour les programmes industriels [8]. En effet :

- Simuler le composant depuis son démarrage nécessite une précision de simulation supérieure à celle offerte par CELTIC, pour simuler les processus bas-niveau (interruptions, co-processeurs). Les intégrer diminuerait la rapidité ou le contrôle du simulateur (voir figure 4.1), et augmenterait sensiblement la complexité de description du composant.
- Certaines variables sont initialisées avant le démarrage du composant, pendant une phase dite de « personnalisation » en usine de la carte. C'est le cas par exemple des clés de chiffrement contenues dans la mémoire non volatile. D'autres variables sont entrées directement par l'utilisateur (comme `g_userPIN`), et doivent donc de toutes façons être fournies par l'utilisateur.
- Certaines hypothèses souhaitables sur le contexte d'appel de la fonction ne sont peut-être pas vraies en simulant directement le programme fourni. Par exemple on peut faire l'hypothèse que l'attaquant peut forger un paramètre de la fonction à une valeur spécifique différent de la valeur « naturelle ».
- Plus généralement, l'injection de fautes peut conduire à lire des emplacements mémoire qui ne sont pas utilisés par l'exécution nominale. Par exemple, une manipulation du flot de contrôle peut amener sur un chemin d'exécution dans une autre fonction qui n'est normalement pas appelée, et donc à lire le contenu d'une variable qui n'est pas initialisée par l'exécution nominale.

Le problème de l'initialisation posé par CELTIC est classique en simulation dynamique. Ce problème a cependant aussi ses avantages : il oblige l'évaluateur à spécifier explicitement les valeurs données aux différentes variables, et par là même les hypothèses formulées sur les entrées. Ceci améliore la répétabilité des expériences par rapport à la revue manuelle, où deux évaluateurs différents peuvent choisir des valeurs différentes sans se concerter.

Implémentation.
Dans CELTIC, l'état initial est précisé dans un script `ruby` permettant de contrôler le cœur `C++` du simulateur. L'évaluateur peut y déclarer des liens entre des noms de variable et des adresses mémoire, puis affecter des valeurs aux différentes variables.

4.2.3. Filtrage des résultats par oracle

La question du traitement des résultats est essentielle dans un outil d'analyse de vulnérabilités, c'est ce qui permet à l'analyse d'être comprise par les évaluateurs. Avant tout, il convient cependant de définir ce qu'on appelle un résultat. Dans CELTIC, on fixe cette réponse grâce aux oracles, qui sont une façon pour les évaluateurs d'exprimer les conditions de succès de l'attaque en lien avec l'état de simulation. Les oracles constituent un formalisme simple à appliquer sur tous les codes sur lesquels on sait les exprimer, et

8. Les exemples de FISSC tiennent compte de ce problème, et proposent une routine simple d'initialisation qui rend la définition de s_0 triviale. Notre exemple « fil rouge » n'a donc en fait pas besoin d'initialisation particulière.

Application	Oracle
`VerifyPIN`	Obtenir le jeton d'authentification avec un mauvais PIN `g_authenticated == 1`
`VerifyPIN`	Ne pas décrémenter le compteur d'essais après un essai de PIN `g_ptc > 2`
`AESKeyCopy`	Forcer une partie de la clé copiée à des valeurs connues `g_key[0] = g_expect[0] || ... || g_key[N-1] = g_expect[N-1]`
`GetChallenge`	Activer le challenge en le forçant à une valeur connue `g_challenge == g_previousChallenge`

`RSA-CRT`	Réussir l'attaque BellCore (voir page 18)
	`(g_cp == pow(m,dp) % p && g_cq != pow(m,dq) % q)`
	`|| (g_cp != pow(m,dp) % p && g_cq == pow(m,dq) % q)`

TABLE 4.3. – Liste d'oracles pour diverses applications

permettent la classification automatique des résultats. Par ailleurs, ils sont configurables par l'utilisateur, contrairement à l'approche suivie par certains des outils spécialisés dans la cryptographie, qui peuvent être tentés de « coder en dur » les conditions de succès des attaques [Fau08 ; And09]. Le principe d'oracle est aussi en phase avec les CC, qui requièrent que les évaluateurs vérifient des scénarios d'attaques préétablis correspondant à l'état de l'art.

Exemple 4.5
On a déjà donné un oracle pour notre exemple fil rouge à la section 4.2.1.3. Ce premier oracle correspond au scénario d'attaque où l'attaquant essaie de s'authentifier avec le mauvais code PIN. On peut proposer un deuxième oracle qui correspond au scénario où le compteur d'essais n'est pas décrémenté correctement après un essai de mauvais code PIN, ce qui permet à l'attaquant de procéder à une recherche exhaustive. On peut écrire cet oracle comme le prédicat : `g_ptc > 2 && g_countermeasure == 0`, en supposant que `g_ptc = 3` dans s_0 et que `g_userPin` y est différent de `g_cardPin`.

Le tableau 4.3 donne une liste de quelques scénarios d'attaque à haut niveau pour diverses applications issues de FISSC, ainsi que des oracles précis construits en utilisant des variables de ces programmes de la même manière que pour notre exemple fil rouge (la condition `g_countermeasure == 0` est implicite et n'est pas rappelée dans les exemples).

Implémentation.
Dans CELTIC, les oracles s'écrivent sous la forme de fonctions booléennes dans le script `ruby` qui contrôle le simulateur. Ces fonctions acceptent l'état de fin de simulation en paramètre, et peuvent l'inspecter pour vérifier la valeur des différentes variables. On dispose donc de toute l'expressivité de `ruby` et de l'ensemble de l'état de simulation pour écrire l'oracle.

4.3. Modèles de faute

Au chapitre 2, on a comparé une liste d'outils à l'état de l'art en fonction des modèles de faute qu'ils supportent. On en a tiré le constat que chaque outil supporte un nombre fixé de modèles de faute.

Dans CELTIC, on vise les objectifs suivants :

— *Permettre aux évaluateurs d'ajouter de nouveaux modèles de fautes.* Ceci est une conséquence du *support de multiples composants*. En effet, on a vu au chapitre 3 que les modèles de faute accessibles dépendaient de l'équipement d'attaque et du composant attaqué. De façon générale, on souhaite que l'outil puisse être étendu dans le futur pour suivre l'évolution technique, tant des contre-mesures matérielles que de l'équipement d'attaque.

— *Supporter les modèles de faute difficiles à mettre en œuvre dans la revue manuelle.* On profite ainsi d'avoir une simulation automatique.

— *Proposer un large choix de modèles de faute.* On aimerait pouvoir proposer au minimum les modèles utilisés par les évaluateurs pendant la revue manuelle, et les modèles classiques décrits dans la littérature.

— *Permettre le support des modèles de fautes probabilistes.* On a défini à la section 3.2.3 du chapitre 3 la notion de modèle de faute probabiliste. Le support de ces modèles particuliers s'entend à la fois **en entrée** de la simulation, pour n'effectuer que des fautes décrites par le modèle, et **en sortie** de la simulation, pour exploiter la probabilité associée aux divers ensembles de fautes et ainsi calculer des métriques de traitement des résultats (voir section 4.4).

Ces objectifs posent un problème similaire à celui de la section 4.2.1.1. Il faut trouver un compromis entre divers critères : on retrouve la rapidité et le contrôle, et l'ergonomie vient remplacer la précision. Par « ergonomie », on entend des modèles faciles à décrire pour l'évaluateur. Le contrôle capte l'expressivité des fautes, c'est-à-dire le problème de savoir quels modèles de faute notre solution permet de couvrir. Enfin, la rapidité capte l'efficacité de la solution, ou plutôt le surcoût en termes de durée d'exécution de l'injection de fautes.

Dans la suite de cette section, on explique comment sont implémentés les fautes et les modèles de fautes, et on discute de l'impact de cette solution sur les trois critères qu'on a présentés.

4.3.1. Faute

On a vu au chapitre 3 qu'une faute pouvait se définir suivant deux espaces duaux : soit en la liant à la perturbation causée par l'équipement, soit en la liant à son effet

sur l'exécution du code. Dans le cadre de CELTIC, c'est ce second cas qui nous intéresse. On cherche à exprimer les modifications, associées à une faute, des opérations qui constituent la sémantique des instructions exécutées par le CPU du composant. Une faute dans CELTIC est donc la donnée d'un quintuplet $(i, \ell, \texttt{op}, v, f)$, où i précise le numéro de l'opération où se produit la faute, ℓ l'emplacement mémoire concerné, `op` le type d'opération (opération de lecture `read` ou d'écriture `write`), v la valeur produite ou utilisée par cette opération, et f la modification de l'opération causée par la faute. Dans CELTIC, on fait le choix de ne pas modéliser de fautes sur les opérations arithmétiques, celles-ci étant couvertes par les fautes sur les opérations de lecture et d'écriture.

4.3.1.1. Type de fautes prédéfinis

Pour des questions d'ergonomie et de performances, CELTIC prédéfinit un certain nombre de types de fautes, permettant de construire les modèles de faute classiques du tableau 2.2 : le modèle de saut de PC, les attaques mémoire volatiles, les attaques registre volatiles, les attaques permanentes.

CELTIC prédéfinit quatre types de fautes, qui permettront d'implémenter les modèles de faute classiques.

1. *Saut de PC, f_{PC}.* La fonction f_{PC} incrémente le registre `PC` d'une taille de saut constante n au moment de l'opération qui consiste à lire le premier mot du code de l'instruction [9].
2. *Attaque mémoire volatile, f_{mem}.* La fonction f_{mem} modifie un nombre n de lectures mémoires consécutives. Un garde vérifie si le nombre d'accès à fauter est strictement positif et si l'emplacement ℓ est un emplacement RAM ou EEPROM, et le cas échéant remplace le résultat v par la valeur fautée, et décrémente le nombre restant d'accès à fauter.
3. *Attaque registre volatile, f_{reg}.* Identique aux attaques mémoire volatiles, mais vise un registre.
4. *Attaque permanente, f_{store}.* Cette f_{store} modifie la valeur contenue dans la mémoire ou un registre en écrivant dans l'état de simulation la valeur fautée à l'emplacement ℓ visé.

4.3.2. Modèle de faute

Le chapitre 3 définit un modèle de faute probabiliste comme plusieurs ensembles de fautes assortis d'une probabilité numérique. Dans CELTIC, on souhaite construire ces ensembles de fautes. Un modèle de faute est dont implémenté comme une fonction qui accepte en entrée une trace d'exécution, et qui retourne la liste des fautes du modèle sur cette trace, en y associant leur probabilité dans le modèle de faute proba-

9. Ce comportement est donc différent d'une faute « NOP » qui remplacerait une instruction par une instruction NOP : si le jeu d'instructions supporte des instructions de tailles variables, un saut de PC peut amener l'interprétation au milieu d'une instruction. Ce n'est pas possible dans le modèle NOP. Cette distinction n'est que rarement relevée dans la littérature.

biliste. On appelle cette fonction la fonction de génération du modèle de faute [10]. Le listing 4.6 donne le pseudo-code d'une fonction de génération qui correspond à l'ensemble de fautes $\{\text{read}_m(EEPROM, a, 0)\}$, c'est-à-dire le remplacement d'une valeur lue dans l'EEPROM par 0.

</>Code source

```
class GZero:
  def generate(xRef):
    attacks = []
    for i, l, op, v in xRef:
      if not l.isMemoryLocation():
        # not a memory location, skip the operation
        continue
      if not l.isEEPROM():
        # not an EEPROM address, skip it
        continue
      if not op.isExec():
        # not fetching an instruction, skip it
        continue
      if v == 0:
        # original value is 0, skip it
        continue
      # add a new f_mem that perturbs the read operation
      # so that it returns 0 instead of v
      attack.push(MemAttack(i, value=0))
```

Listing 4.6. – Fonction de génération d'un modèle de mise à zéro d'une lecture EEPROM (pseudo-code)

On peut implémenter les modèles classiques présentés dans la table 2.2 du chapitre 2 comme des fonctions de génération à l'aide des fautes prédéfinies :

— *InstSkip*. Il suffit de parcourir la trace d'exécution en générant des f_{PC} à chaque opération correspondant au début du décodage d'une instruction.
— *InstRepl*. On peut générer des remplacements d'instruction en faisant des fautes f_{mem} sur la mémoire qui correspond au code afin de remplacer la valeur normalement lue par toutes les autres valeurs possibles.
— *BranchChoice*. Ce modèle est un cas particulier de *InstRepl*. À chaque instruction de branchement conditionnel, on remplace l'instruction décodée par une instruction opposée (via des f_{mem} sur les lectures des codes opération).
— *Assign*. On utilise des f_{store} pour assigner la valeur souhaitée aux emplacements

10. Cette fonction est analogue à la fonction caractéristique de l'ensemble qui génère tous les éléments de l'ensemble. Cette fonction n'est pas toujours calculable. Bien entendu, dans CELTIC, on se limite aux cas où elle l'est.

mémoires correspondants à la variable visée.
— *DataCorrupt.* Les corruptions volatiles sont couvertes en générant des f_{mem}, les corruptions permanentes sont couvertes avec des f_{store} sur la mémoire.
— *RegCorrupt.* Identique à *DataCorrupt*, mais avec des fautes sur les registres.

Il est également possible de composer des modèles de faute dans CELTIC, c'est-à-dire de construire des modèles qui renvoient les fautes générées par plusieurs sous-modèles. Pour ce faire, on définit la fonction de génération comme retournant la concaténation des attaques retournées par les fonctions de génération des modèles à composer (voir exemple 4.6).

Exemple 4.6 *Composition de modèles de faute*
Soit le modèle $\mathcal{M}_{0,1}$ composé des deux ensembles de fautes : $\{\text{read}_m(EEPROM, a, 0)\}$ et $\{\text{read}_m(EEPROM, a, 1)\}$, pour a quelconque non nul. Le modèle $\mathcal{M}_{0,1}$ peut s'implémenter comme la composition des deux fonctions de générations g_0 et g_1. Le code 4.7 donne la fonction de génération du modèle $\mathcal{M}_{0,1}$ qui utilise les deux fonctions de génération g_0 et g_1. Le code 4.6 donne la fonction de génération pour g_0.

</>Code source

```
class MZeroOne:
  def generate(xRef):
    attacks = []
    # concatenate attacks from g_zero
    attacks = attacks + GZero.generate(xRef)
    # concatenate attacks from g_one
    attacks = attacks + GOne.generate(xRef)
```

Listing 4.7. – Fonction de génération d'un modèle $M_{0,1}$ (pseudo code)

4.3.3. Discussion

La solution proposée en termes de modèle de faute atteint-elle les objectifs définis en début de section ? On discute de chaque objectif :
— *Permettre aux évaluateurs d'ajouter de nouveaux modèles de fautes.* On a vu que pour CELTIC, un modèle de faute s'implémente comme une fonction de génération. Il est possible pour l'évaluateur d'en rajouter de nouveau via des scripts `ruby`.
— *Supporter les modèles de faute difficiles à mettre en œuvre dans la revue manuelle.* Les modèles de faute difficiles à mettre en œuvre le sont pour deux raisons : (1) soit parce qu'ils font intervenir l'encodage binaire, sur lequel il est difficile de raisonner manuellement, (2) soit parce qu'ils font intervenir une combinatoire importante

(ces deux raisons ne s'excluent pas). L'apport de CELTIC est net pour (1), il est même plutôt plus simple de manipuler l'encodage binaire dans les modèles de fautes de l'outil. Pour (2) en revanche, les modèles de fautes combinatoires génèrent un nombre important de fautes, qu'il est donc très long de simuler [11]. Dans ce contexte, on recommande de commencer par tester des modèles rapides (ceux de la revue manuelle par exemple, le saut d'instruction, etc.) puis d'utiliser des modèles plus complets seulement dans un second temps, par exemple si les modèles rapides n'ont rien donné.

— *Proposer un large choix de modèles de faute.* On a montré que CELTIC couvre les modèles classiques proposés dans la littérature. Par ailleurs, l'outil propose une méthode pour les composer facilement.

— *Permettre le support des modèles de fautes probabilistes.* Le modèle de faute probabiliste est implémenté comme la composition de fonctions de génération, qui associent également une probabilité aux fautes générées.

4.4. Métriques de traitement des résultats

On a vu à la section 4.2.3 que CELTIC est capable de filtrer automatiquement les résultats à l'aide d'oracles. Dès qu'on s'éloigne un peu des modèles de fautes simples à appliquer en revue manuelle tels que le NOP (voir exemple 4.7), cela n'est cependant plus suffisant pour offrir une analyse utile pour les évaluateurs. Dans bien des cas, il est nécessaire de traiter les résultats obtenus. En effet, les résultats bruts posent plusieurs problèmes :

— Ils ne quantifient pas la sécurité globale de l'application. Pour les évaluateurs, une telle métrique est nécessaire pour émettre un diagnostic sur l'application. Ce type de métrique peut également servir à comparer plusieurs implémentations entre elles. On appelle ces métriques des métriques globales.

Définition 4.1 *Métrique globale*
Quantité numérique calculable qui représente le degré de vulnérabilité global d'une application aux attaques par injection de fautes.

— On ne sait pas quantifier la dangerosité individuelle des fautes découvertes. Or, c'est une nécessité dans le cadre des CC, pour décider si les attaques sont « réalistes » ou pas suivant le contexte (puissance de l'attaquant, voir la table 1.3 du chapitre 1). Par ailleurs, on sait identifier une attaque réussie grâce aux oracles, mais obtenir les caractéristiques de la faute associée ne suffit pas à comprendre *comment* elle met en danger la sécurité. Cette métrique serait pourtant utile pour comprendre les résultats de l'outil (éventuellement, les valider) et surtout savoir comment protéger le code [12]. On appelle une telle métrique une métrique locale.

11. Le temps de simulation est quadratique en le nombre d'instructions, car le nombre de fautes générées (et donc, de simulations) est proportionnel au nombre d'instructions, et la durée d'une simulation est également proportionnelle en le nombre d'instructions.

12. Le choix des contre-mesures pour protéger une implémentation est une problématique qui

Définition 4.2 *Métrique locale*
Ensemble de quantités numériques calculables, chacune étant liée à un paramètre d'attaque du code, et représentant le degré de vulnérabilité de l'application pour ce paramètre.

Dans le cadre de la thèse on a donc défini une métrique globale, le taux de vulnérabilité $\mathcal{V}$, et sa variante locale, le taux de sensibilité $\mathcal{S}_\ell$ associé à l'emplacement mémoire ℓ (registre, adresse EEPROM, etc.). Le reste de cette section présente brièvement les métriques décrites dans la littérature, définit en détail $\mathcal{V}$ et $\mathcal{S}_\ell$, et offre une évaluation expérimentale de $\mathcal{V}$.

4.4.1. État de l'art des métriques globales

L'état de l'art du traitement des résultats a déjà été couvert en partie à la section 2.2.5. Ici, on détaille les métriques existantes dans la littérature, en commençant par les métriques globales, puis les métriques locales. On présente la métrique « naturelle » obtenue expérimentalement à partir d'attaques sur carte, puis la métrique N, le nombre de fautes réussies, et le taux de réussite τ qui est le rapport du nombre d'attaques réussies sur le nombre de fautes effectuées.

4.4.1.1. Métrique « naturelle » φ

En dehors de tout outil, les évaluateurs utilisent traditionnellement les tests de pénétration pour déterminer le taux de réussite d'une attaque, défini comme :

$$\varphi = \frac{\#\text{ of experimentally successful attacks}}{\#\text{ of experimentally injected faults}}$$

le rapport du nombre d'attaques réussies sur le nombre de fautes injectées expérimentalement. La métrique « naturelle » ne pose pas de problèmes de réalisme, puisque ce sont les résultats des véritables expérimentations physiques. L'expérimentation physique requiert de l'équipement et de l'expertise, ainsi que du temps : de l'ordre d'une semaine pour une expérience, sans compter la mise en place du matériel pour attaquer la carte et la réduction de l'espace des paramètres d'attaques. Par ailleurs chaque modification dans l'implémentation force à recommencer ce processus coûteux.

4.4.1.2. Nombre d'attaques réussies N

Une première métrique globale sur le code est le nombre d'attaques réussies N. Parmi les outils présentés au chapitre 2, cette métrique est utilisée par [Pot+14] et [Ber+14]. Dans sa thèse [Mor14], Moro relève que cette métrique se comporte comme un « pire cas » qui force à tenir compte de toutes les fautes possibles. L'exemple 4.7 montre une application de la métrique N sur un modèle de faute « complexe » (c'est-à-dire difficile à appliquer en revue de code manuelle).

concerne surtout les développeurs, mais pouvoir fournir des explications sur l'origine d'une vulnérabilité peut être considéré comme faisant partie de la valeur ajoutée d'un CESTI.

Fonction `byteArrayCompare`	
Adresse	Nombre d'attaques réussies
0x41ab	1
0x41ac	1
0x41ad	2
0x41ae	5
0x41b2	4
0x41b6	4
0x41b8	1
0x41b9	1
0x41ba	12
0x41bb	1
0x41bc	3
0x41bd	5
0x41be	7
0x41c1	1
0x41c2	2
0x41c8	1
0x41c9	61
0x41cb	8
0x41cc	1
0x41cd	6
0x41cf	115

Fonction `VerifyPIN`	
Adresse	Nombre d'attaques réussies
0x41da	1
0x41db	1
0x41dd	1
0x41df	1
0x41e5	3
0x41e6	2
0x41e7	1
0x41e8	1
0x41e9	8
0x41ea	1
0x41eb	10
0x41f0	1
0x41f1	4
0x41f3	8
0x41f5	1
0x41fb	6
0x41fd	136
0x4210	2
0x4214	2

TABLE 4.4. – Résultats avec un modèle de faute complexe

Exemple 4.7 *Résultats avec un modèle de faute complexe*
On a effectué une analyse sur notre exemple fil rouge avec le modèle de faute de remplacement exhaustif d'un octet du code et l'oracle d'authentification. Ce modèle est difficile à appliquer dans la revue manuelle, car il dépend de l'encodage binaire des instructions, et car il existe beaucoup de faute possibles (255 pour chaque octet de code binaire). On donne dans la table 4.4 le nombre de fautes par adresse qui conduisent à des attaques réussies. Cette représentation est suffisante pour déterminer que les trois instructions vulnérables du modèles de fautes NOP sont également vulnérables avec ce modèle (l'instruction à l'adresse `0x41c8` est ainsi vulnérable une fois sur son opérande en `0x41c8` et 61 fois sur son code opération à l'adresse `0x41c9`), mais on constate d'autre attaques réussies (par exemple, 8 en `0x41e9`), qu'il semble plus difficile d'expliquer à première vue. Il s'agit d'attaques qui dépendent de l'encodage binaire, et qui sont donc spécifiques à ce modèle de faute, et difficile à trouver via la revue manuelle. Pour le lecteur intéressé, on rassemble les explications des attaques marquées en **gras** de la table 4.4 dans l'annexe C.

Un inconvénient de la métrique N est sa sensibilité au paradoxe de la surface d'at-

taque, qu'on définit ci-dessous.

Définition 4.3 *Paradoxe de la surface d'attaque*
Les contre-mesures sur un code ont souvent pour effet secondaire d'augmenter la taille du code. De ce fait, la surface d'attaque, c'est-à-dire le nombre de points où injecter une faute est possible, augmente avec l'ajout de contre-mesures, et le nombre d'attaques réussies remontées par les outils a également tendance à augmenter. Ceci est paradoxal, car les contre-mesures ont pour but d'augmenter la sécurité du code, donc devraient avoir pour effet de diminuer la métrique de vulnérabilité globale du code [a].

a. L'expertise des CESTI et des développeurs donne une confiance suffisante concernant l'efficacité de certaines contre-mesures

Ainsi, si un code protégé multiplie sa longueur par 10 et le nombre d'attaques réussies par seulement 1.5, cette métrique donne inconditionnellement le code protégé comme plus vulnérable que celui d'origine.

4.4.1.3. Taux de succès τ

Une alternative qui semble résoudre le paradoxe de la surface est de tenir compte du nombre total d'attaques effectuées dans le calcul :

$$\tau = \frac{\#\text{ of successful attacks}}{\#\text{ of injected faults}}$$

τ est similaire dans son principe à la métrique « naturelle ». Comme elle, elle donne une probabilité de succès d'une attaque par fautes. Elle semble également résoudre le paradoxe de la surface, puisqu'un code avec une plus grande surface d'attaque permet d'effectuer davantage d'attaques, ce qui augmente le dénominateur. En revanche, cette métrique crée une nouvelle incohérence, relevée dans le cadre de la tolérance aux fautes par Schirmeier *et al.* [SCS15] : si on se munit par exemple du modèle de faute qui remplace une instruction par un NOP, alors la contre-mesure, baptisée *Dilution Fault-Tolerance mechanism* dans [SCS15], qui consiste à rajouter des instructions NOP autour d'un code à protéger, est efficace. Pour chaque instruction NOP ajoutée, une nouvelle faute est possible ce qui augmente de 1 le dénominateur de τ, mais cette faute ne peut pas réussir puisque transformer un NOP en un NOP est... un NOP. En utilisant cette technique, il est possible de rendre τ arbitrairement petit (aux contraintes de longueur du code près). Pourtant, cette « contre-mesure » n'augmente pas la sécurité du code en pratique : il est facile pour un attaquant de constater qu'il est inutile d'attaquer la zone comportant les NOP [13].

13. On notera que, en l'absence d'un attaquant, Schirmeier *et al.* justifient différemment en quoi cette technique de dilution n'améliore pas la sûreté : ils considèrent comme impossible de comparer des nombres d'erreurs sur des espaces de fautes de tailles différentes.

Définition 4.4 *Paradoxe de dilution*
Toute technique qui vise à augmenter le nombre de fautes injectées sans modifier le nombre d'attaques réussies a pour effet mécanique de diminuer τ, indépendamment de l'impact réel sur la sécurité. On appelle cet inconvénient de la métrique τ le paradoxe de dilution.

4.4.2. Taux de vulnérabilité $\mathcal{V}$

Dans cette section, on définit $\mathcal{V}$, une métrique globale conçue pour répondre aux problèmes actuellement posés par les métriques existantes :
— Ne pas souffrir du paradoxe de l'augmentation de la surface.
— Ne pas souffrir du paradoxe de dilution.
— Tenir compte des différentes probabilités d'occurrences des fautes.

Par ailleurs, on souhaite pouvoir utiliser cette métrique pour estimer le temps nécessaire à l'identification et à l'exploitation de l'attaque. Ce facteur *elapsed time* est une des données requises dans le processus CC pour évaluer le niveau d'`AVA_VAN` auquel le composant peut prétendre (voir tableau 1.3 de la section 1.1.3).

Pour répondre à ces objectifs, on propose d'utiliser les modèles de fautes probabilistes tels que définis dans le chapitre 3.

On se munit d'une définition initiale « idéale » pour la métrique proposée :

$$\mathcal{V} = \Pr(\text{Attaque réussie}) \tag{4.1}$$

L'équation (4.1) définit ainsi $\mathcal{V}$ comme la probabilité qu'une attaque par injection de faute réussisse.

Dans le cadre d'un modèle de faute probabiliste, on se munit d'un espace de paramètres d'équipement $\mathcal{P}$. Chaque élément p de cet espace $\mathcal{P}$ correspond à des valeurs précises des paramètres de l'équipement d'attaque.

Mathématiquement, l'équation (4.1) peut alors s'écrire :

$$\mathcal{V} = \sum_{p \in \mathcal{P}} \Pr(\text{Attaque réussie} \mid p) \cdot \Pr(p) \tag{4.2}$$

Dans l'équation (4.2), $\Pr(\text{Attaque réussie} \mid p)$ est la probabilité qu'une attaque réussisse sachant que l'équipement est réglé avec la valeur de paramètres $p \in \mathcal{P}$. L'équation (4.2) exprime donc que la probabilité globale de réussite d'une attaque est égale à la somme sur tous les paramètres de l'espace du produit d'une probabilité de réussite suivant une valeur des paramètres p fixé avec la probabilité que les paramètres d'équipement aient bien pour valeur p. Par définition des paramètres d'équipement, leur valeur est contrôlable par l'attaquant. La probabilité $\Pr(p)$ peut donc s'interpréter comme la probabilité que l'attaquant choisisse cette valeur p de paramètre. La modélisation du choix de l'attaquant, et donc le calcul de la probabilité $\Pr(p)$ est détaillé à la section 4.4.3.

Pour pouvoir calculer $\mathcal{V}$ à partir de l'équation (4.2), on doit préciser ce qu'on entend par attaque réussie : dans le cadre de la thèse, cela signifie que la faute injectée par l'attaque amène à vérifier l'oracle pour l'exécution (que cette dernière soit produite par

simulation ou réelle sur carte). Or, une des sorties de CELTIC est précisément la liste $\mathcal{F}_\mathcal{S}$ des fautes qui conduisent à une attaque vérifiant l'oracle. En utilisant l'espace de fautes $\mathcal{F}_\mathcal{S}$, l'équation (4.2) peut alors s'écrire :

$$\mathcal{V} = \sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_\mathcal{S}} \Pr(F = f \mid p) \cdot \Pr(p) \tag{4.3}$$

L'équation (4.3) exprime donc que le taux de vulnérabilité est égal à la somme des probabilités d'obtenir une des fautes de $\mathcal{F}_\mathcal{S}$, modulée par les choix des valeurs de paramètres d'équipement. Or, $\Pr(F = f \mid p)$ est donné par le modèle de faute probabiliste [14] (voir section 3.2.3). $\mathcal{V}$ est donc calculable sous cette forme, à condition de savoir modéliser $\Pr(p)$, ce qui est l'objet de la section suivante.

4.4.3. Modèle d'attaquant

La métrique $\mathcal{V}$ proposée déplace le problème de la surface d'attaque vers la répartition des probabilités entre les différents $\Pr(p)$.

Définition 4.5 *Modèle d'attaquant*
Ensemble de moyens de construction des lois de probabilités associées à $\Pr(p)$.

La modélisation choisie a un fort impact sur la valeur de $\mathcal{V}$. Dans la suite, on propose plusieurs modèles utiles.

4.4.3.1. Modèle d'attaquant omniscient

Un attaquant omniscient connaît la valeur précise p_{max} des paramètres telle que

$$\forall p \in \mathcal{P}, \sum_{f \in \mathcal{F}_\mathcal{S}} \Pr(F = f \mid p_{max}) \geqslant \sum_{f \in \mathcal{F}_\mathcal{S}} \Pr(F = f \mid p)$$

Il est alors capable de maximiser le taux de vulnérabilité pour la carte en n'attaquant qu'au paramètre p_{max}. $\mathcal{V}$ se calcule alors comme :

$$\mathcal{V} = \sum_{f \in \mathcal{F}_\mathcal{S}} \Pr(F = f \mid p_{max})$$

Ce modèle est « idéal », dans le sens où un attaquant ne connaît jamais p_{max} en pratique. Ce modèle est néanmoins utile pour calculer une borne supérieure de $\mathcal{V}$, dans une démarche similaire à ce que proposent Moro *et al.* en assimilant le nombre d'attaques réussies à un « pire cas ».

14. Pour être précis, on a vu qu'un modèle probabiliste donne $\Pr(C = c \mid p)$, pour c un paramètre de code. Dans ce chapitre, on considère les « fautes » au sens de leur effet sur le code, d'où le changement de notation.

4.4.3.2. Attaquant à connaissance nulle

À l'inverse, un attaquant qui n'a aucune connaissance de l'application ou de sa plate-forme ne peut privilégier aucun paramètre et doit donc appliquer une stratégie uniforme en assignant à chaque $\Pr(p)$ la valeur $\frac{1}{|\mathcal{P}|}$. Ceci correspond à un parcours « à l'aveugle » des différentes valeurs possibles des paramètres d'équipement. Dans ce scénario, le calcul de $\mathcal{V}$ devient alors :

$$\mathcal{V} = \frac{\sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{\mid \mathcal{P} \mid}$$

4.4.3.3. Attaquant réaliste

Bien souvent, un attaquant « réaliste » a une position intermédiaire entre ces deux extrêmes : il est capable de réduire l'espace des paramètres à tester suivant certains éléments d'observation des canaux cachés. Par exemple, l'attaquant peut déterminer qu'après une certain point, le code ne s'occupe plus que des entrées/sorties : il est inutile pour lui d'attaquer avec des paramètres qui perturbent l'exécution passé ce point. L'attaquant va donc pouvoir déterminer un sous-espace $\mathcal{P}'$ de paramètres sur lequel mener ses attaques. À l'intérieur de ce sous-espace, la stratégie appliquée peut être similaire à celle d'un attaquant à connaissance nulle (si aucune information ne permet de distinguer les valeurs entre elles), mais des distinctions plus subtiles sont possibles (par exemple, subdiviser en deux sous-espaces et concentrer 2/3 des attaques sur l'un des deux). Le calcul de $\mathcal{V}$ dans un sous-espace $\mathcal{P}'$ est alors :

$$\mathcal{V} = \frac{\sum_{p \in \mathcal{P}'} \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{\mid \mathcal{P}' \mid}, \text{ pour } \mathcal{P}' \subset \mathcal{P}$$

On constate qu'un attaquant réaliste ne souffre pas du paradoxe de la surface, tout en étant insensible au paradoxe de dilution. En effet, l'augmentation de la surface d'attaque, si elle correspond à un durcissement du code, oblige l'attaquant à tester le code supplémentaire, et donc tend à diminuer le taux de vulnérabilité. À l'inverse, la dilution par ajout de NOP autour du code « utile » créée du code qu'il est facile d'identifier comme insensible aux attaques, et donc d'exclure de l'analyse [15].

Les différences drastiques entre modèles d'attaquant signifient qu'il est nécessaire de préciser le modèle utilisé dans le calcul d'un taux de vulnérabilité.

4.4.3.4. Calcul pratique pour un attaquant réaliste

Dans la pratique, le calcul des $\Pr(p)$ n'est pas trivial dans le cas d'un modèle attaquant réaliste. La difficulté est de déterminer l'espace des paramètres d'équipement $\mathcal{P}$. Pour un évaluateur qui effectue les tests de pénétration sur carte, on peut **dans la pratique**

15. De façon intéressante, ceci signifie également que des instructions NOP ajoutées de façon à ce que l'attaquant ait du mal à les discerner des instructions utiles (par exemple, un petit nombre aléatoire de NOP entre deux instructions utiles) peut réellement contribuer à améliorer la sécurité et donc diminuer le taux de vulnérabilité, si on suppose que cette modification n'augmente pas le nombre d'attaques réussies.

décrire l'espace suivant une dimension temporelle $\mathcal{P}_T$ et une dimension spatiale $\mathcal{P}_S$. Le cardinal de $\mathcal{P}$ se calcule donc comme le produit des cardinaux de $\mathcal{P}_T$ et $\mathcal{P}_S$. Le cardinal de $\mathcal{P}_S$ est égal au nombre de positions S de l'équipement considérées pendant l'attaque, généralement une seule position jugée optimale une fois les tests d'inférence de modèle effectués. Le cardinal de $\mathcal{P}_T$ dépend de 3 paramètres : les temps t_0 et t_1 qui délimitent respectivement la première et la dernière position temporelle, et le pas δ discret entre 2 positions temporelles. Dès lors, le cardinal de $\mathcal{P}_T$ est simplement :

$$| \mathcal{P}_T |= \frac{t_1 - t_0}{\delta} \tag{4.4}$$

On obtient alors :

$$| \mathcal{P} |=| \mathcal{P}_S | \cdot | \mathcal{P}_T |= S \frac{t_1 - t_0}{\delta} \tag{4.5}$$

Néanmoins, l'équation (4.5) n'est pas utilisable par un évaluateur qui ne souhaite pas procéder à l'expérience sur carte : au minimum, il faut lancer le code sur carte pour déterminer sa durée et ainsi t_0 et t_1. Il est pourtant crucial de permettre ce scénario, car évaluer le taux de vulnérabilité sans expérience physique est un des avantages majeurs de la méthode que nous proposons.

Pour rendre ce scénario possible, on propose d'approximer la durée $t_1 - t_0$ en fonction du nombre c d'instructions exécutées dans l'intervalle d'attaque choisi. Plus précisément, on cherche α tel que :

$$t_1 - t_0 \approx \alpha\, c \tag{4.6}$$

où α exprime la durée d'une instruction (unité : nanosecondes par instruction). Cette approximation est raisonnable car la plupart des composants modernes garantissent un nombre constant de cycles d'horloge par seconde. Il existe cependant plusieurs sources d'imprécisions :

— Toutes les instructions ne durent pas nécessairement le même nombre de cycles.
— Les branchements dans le code ont tendance à ralentir l'exécution. De nombreux algorithmes (prédictions de branches par exemple) visent à diminuer le coût du branchement, mais leurs implémentations matérielles dans les processeurs sont rarement disponibles et leurs performances difficiles à modéliser.
— Certaines contre-mesures matérielles interviennent sur la fréquence d'horloge en la rendant légèrement irrégulière (*jitter*).
— Certaines opérations matérielles (écritures de mémoire non volatile, calculs cryptographiques matériels) prennent un temps plus long et difficile à déterminer à partir des seules instructions.

On pourra néanmoins se contenter en général de cette approximation car les codes étudiés sont de taille modeste. Pour la carte C, on a calculé $\alpha = 117$ à partir de la durée et du nombre d'instructions du programme de détection de fautes. On a ensuite comparé dans le tableau 4.5 les temps obtenus avec $\alpha\, c$ sur plusieurs applications, avec les temps réels observés. Sur ces exemples, on retrouve le même α entre 10 et 15% près. Les imprécisions dans ce cas s'expliquent par la mesure approximative de la durée du code (on est à 150000 ± 1000ns) et par la structure de boucle du code de détection (les sauts fréquents diminuent légèrement le nombre d'instructions exécutées par seconde).

Programme	Durée $t_1 - t_0$ (ns)	Nombre d'instructions c	$\frac{t_1-t_0}{c}$	Erreur
Référence	150000	1282	117	0
`VerifyPIN_2`	38000	296	128	9%
`VerifyPIN_5`	40000	295	135	15%
`VerifyPIN_7`	44000	345	128	9%

TABLE 4.5. – Comparaisons de α

En définitive, on propose la méthode de calcul suivante pour $\mathcal{V}$ dans le cas d'un attaquant uniforme sur un intervalle réaliste :

Méthode de calcul de $\mathcal{V}$ pour un modèle d'attaquant réaliste :

1. Une fois par composant sécurisé, effectuer la méthode d'inférence de modèle de faute présentée au chapitre 2. En tirer δ (le pas entre deux paramètres temporels), S (le nombre de positions spatiales de l'équipement), α (la durée d'une instruction), et le modèle de faute probabiliste $\mathcal{M}_{d,e}$.
2. Configurer CELTIC pour effectuer le *golden run* sur l'application à tester.
3. Déterminer un intervalle réaliste d'instructions à cibler pour l'attaquant (exclure les NOP et autres attentes actives, les routines qu'il est facile d'identifier comme de l'Entrée/Sortie, etc.). En tirer c, le nombre d'instructions exécutées dans la trace du *golden run* sur cet intervalle.
4. Utiliser CELTIC avec $\mathcal{M}_{d,e}$ sur l'intervalle choisi pour déterminer $\mathcal{F}_S$.
5. Calculer $\mathcal{V}$ dans le modèle d'attaquant réaliste uniforme à partir de $\mathcal{M}_{d,e}$, α, δ et $\mathcal{F}_S$:

$$\mathcal{V} = \delta \frac{\sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{S\,\alpha\,c} \tag{4.7}$$

Si la valeur de α est imprécise, on constate que $\mathcal{V}$ sera imprécis à une constante de proportionnalité près, ce qui ne fausse pas en général complètement le résultat du calcul (en particulier, une implémentation plus robuste aura bien un taux de vulnérabilité plus faible qu'une implémentation moins robuste).

Implémentation.
CELTIC propose le calcul de $\mathcal{V}$ dans le modèle d'attaquant réaliste après une campagne d'injection de faute. L'utilisateur doit préciser dans le script les paramètres S, δ, α et c (ou directement $\mid \mathcal{P} \mid$), le calcul des $\Pr(F = f \mid p)$ est automatique en utilisant les probabilités attachées au modèle de faute choisi. Le résultat du calcul est reporté dans un fichier qui contient aussi la liste des attaques réussies.

4.4.4. Évaluation de la métrique proposée

On a comparé notre métrique globale $\mathcal{V}$ avec la métrique classique τ (définie à la section 4.4.1.3), en utilisant la métrique « naturelle » φ comme référence. Ces comparaisons

Card	Command	$\mathcal{V}$	τ	φ	$\mid \mathcal{P}' \mid$
A	`VerifyPIN`	2.35×10^{-5}	3.2×10^{-2}	3.40×10^{-5}	5883
A	`SecureVerifyPIN`	2.08×10^{-6}	8.5×10^{-5}	0	5000
A	`GetChallenge`	2.01×10^{-5}	1.75×10^{-3}	2.94×10^{-5}	6800
A	`SecureGetChallenge`	7.1×10^{-7}	2.74×10^{-6}	0	3000
B	`GetChallenge`	1.1×10^{-3}	1.2×10^{-3}	1.4×10^{-3}	231
B	`SecureGetChallenge`	0	2.14×10^{-4}	0	833

TABLE 4.6. – Comparaisons de métriques globales

portent sur plusieurs implémentations de `VerifyPIN` et `GetChallenge` à deux niveaux de protections distincts : une version naïve sans protection et une version plus sécurisée avec des contre-mesures. Le tableau 4.6 rassemble les résultats sur ces diverses implémentations sur deux cartes : la carte A sécurisée avec un processeur Cortex-M (ARMv7-M) et la carte B sécurisée avec un processeur CISC propriétaire. La carte A subit des attaques électromagnétiques tandis que la carte B subit des attaques laser. La métrique $\mathcal{V}$ a été calculée avec des espaces de paramètres dont les cardinaux sont donnés dans la colonne $\mid \mathcal{P}' \mid$ sur les modèles de faute probabilistes obtenus par inférence de modèle de faute pour les cartes A et B (modèles présentés à la section 3.5.1 du chapitre 3). La métrique τ a été calculée à partir d'un modèle de fautes « générique » de remplacement d'instruction (*InstRepl*). Ces deux métriques ont été déterminées à l'aide des attaques réussies renvoyées par CELTIC dans leur modèle de faute respectif. La métrique φ a été déterminée à partir des attaques réussies sur carte pendant des expérimentations de trois jours chacune.

Pour toutes les implémentations naïves, $\mathcal{V}$ est de même ordre de grandeur que φ. Pour toutes les implémentations sauf la version naïve de `GetChallenge` sur carte B, τ est bien supérieure à φ. Cette différence peut s'expliquer par les incohérences entre le modèle de faute choisi arbitrairement et les fautes réellement injectables sur carte. Dans la version sécurisée de `GetChallenge` sur la carte B, on trouve $\mathcal{V} = \varphi = 0$, tandis que pour les autres implémentations sécurisées, on a $\varphi = 0 \neq \mathcal{V}$. Ces différences signifient que CELTIC muni du modèle de faute probabiliste prévoit des vulnérabilités théoriques, mais qu'aucune ne s'est réalisée en pratique pendant l'expérience sur carte. Trois hypothèses permettent d'expliquer ce phénomène :

— Certaines fautes obtenues pendant l'inférence du modèle de faute dépendent de paramètres cachés (éventuellement non contrôlables) dont la valeur durant l'expérience n'ont pas permis d'obtenir la faute souhaitée.
— La phase de généralisation du modèle de faute a ajouté des fautes qui ne peuvent pas se produire en pratique avec les paramètres considérés.
— Le fait d'injecter une certaine faute étant probabiliste, l'expérience n'a simplement pas duré suffisamment longtemps pour observer l'événement.

En tout état de cause, la précision de $\mathcal{V}$ reste meilleure que τ sur ces exemples.

4.4.4.1. Correspondance avec l'*elapsed time*

Il est possible d'établir une correspondance entre le facteur *elapsed time* du potentiel d'attaque et $\mathcal{V}$. En effet, d'après l'équation (4.1), $\mathcal{V}$ représente la probabilité de réussite d'une attaque par perturbation quelconque. On peut voir le fait d'obtenir une attaque

Card	Command	$(s \times \mathcal{V})^{-1}$ (ET)	$(s \times \tau)^{-1}$ (ET)	$(s \times \varphi)^{-1}$ (ET)
A	`VerifyPIN`	8h (3)	24s (0)	6h (3)
A	`SecureVerifyPIN`	1w (4)	2.5h (3)	$>$ *3d* ($\geqslant$ 4)
A	`GetChallenge`	10h (3)	7min (0)	7.4h (3)
A	`SecureGetChallenge`	2w (6)	3.5d (4)	$>$ *3d* ($\geqslant$ 4)
B	`GetChallenge`	5min (0)	5min (0)	5min (0)
B	`SecureGetChallenge`	not practical (_)	20min (3)	$>$ *3d* ($\geqslant$ 4)

TABLE 4.7. – Comparaison de prédictions d'*elapsed time*

Elapsed Time	Exploitation Rating
$<$ one hour	0
$<$ one day	3
$<$ one week	4
$<$ one month	6
$>$ one month	8
Not practical	_

TABLE 4.8. – Cotations de l'*elapsed time* dans le potentiel d'attaque

.

réussie comme un test de Bernouilli, dont la probabilité de succès est $\mathcal{V}$. L'espérance du nombre d'attaques nécessaires pour obtenir un succès est donc $\frac{1}{\mathcal{V}}$. Si l'on dispose du nombre d'attaques s réalisées par l'équipement d'attaque par seconde, alors on peut calculer un temps $t_{\mathcal{V}}$:

$$t_{\mathcal{V}} = \frac{1}{s\,\mathcal{V}} \tag{4.8}$$

À partir des résultats de la table 4.6, on calcule ainsi les temps $t_{\mathcal{V}}$ qu'on compare avec $t_\tau = \frac{1}{s\,\tau}$ et $t_\varphi = \frac{1}{s\,\varphi}$. t_φ est la mesure de temps « naturelle » utilisée par les évaluateurs pour calculer le facteur *elapsed time*.

La table 4.7 rassemble et compare les temps obtenus. Pour chaque temps obtenu, la cotation correspondante dans la JIL du potentiel d'attaque est indiquée entre parenthèses. La signification de chaque cotation est explicitée dans la table 4.8. Ces résultats confirment ceux obtenus en utilisant directement les métriques (ce qui n'est pas très surprenant puisqu'ils sont issus d'un produit par le même facteur), en les précisant un peu : $\mathcal{V}$ prévoit une attaque réussie après une ou deux semaines pour les implémentations sécurisées, alors que les attaques sur carte n'ont duré que trois jours : il est donc tout à fait probable que les expérimentations n'aient pas permis de mettre en évidence les vulnérabilités existantes (cependant les autres hypothèses explicatives restent possibles). Ces exemples confirment que la métrique $\mathcal{V}$ peut être utilisée pour prédire le temps nécessaire à l'obtention d'une attaque réussie sur carte.

4.4.4.2. $\mathcal{V}$ en l'absence de modèle de faute probabiliste

En l'absence d'un modèle probabiliste (modèle non probabiliste « générique »), il manque l'information : $\Pr(F = f \mid p)$. Par ailleurs, si le modèle générique n'est pas associé à un équipement d'attaque il n'est pas possible de décrire $\mathcal{P}$. On perd donc en qualité de comparaison avec φ et la prédiction de l'*elapsed time*.

Dans cette situation, il est possible d'exprimer la métrique classique τ à l'aide de $\mathcal{V}$. Pour ce faire, on se munit d'un modèle de faute non probabiliste M, de la liste $\mathcal{F}$ des fautes effectuées, ainsi que de la liste $\mathcal{F}_S$ des fautes conduisant à une attaque réussie retournée par CELTIC sur un programme à évaluer avec le modèle M. Comme le modèle n'est pas probabiliste, on considère que l'attaquant peut choisir la faute $f \in \mathcal{F}$ à réaliser en choisissant le paramètre d'équipement p. Pour chaque faute f, on a donc un unique p_f tel que : $\Pr(F = f \mid p_f) = 1$ et $\forall f' \neq f, \Pr(F = f' \mid p_f) = 0$. On définit $\mathcal{P} \hat{=} \bigcup_{f \in \mathcal{F}} p_f$. Dans ces conditions, on obtient :

$$\begin{aligned}\mathcal{V} &= \sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p) \cdot \Pr(p) \\ &= \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p_f) \cdot \Pr(p_f)\end{aligned}$$

Or, $\Pr(F = f \mid p_f) = 1$. On a alors :

$$\mathcal{V} = \sum_{f \in \mathcal{F}_S} \Pr(p_f)$$

En supposant que l'attaquant ne favorise pas une faute plutôt qu'une autre, on a $\Pr(p_f) = \mid \mathcal{P} \mid^{-1}$. Par définition de $\mathcal{P}$, on a donc : $\Pr(p_f) = \mid \mathcal{F} \mid^{-1}$.
On obtient finalement :

$$\begin{aligned}\mathcal{V} &= \frac{\sum_{f \in \mathcal{F}_S} 1}{\mid \mathcal{F} \mid} \\ &= \frac{\mid \mathcal{F}_S \mid}{\mid \mathcal{F} \mid} \\ &= \tau\end{aligned}$$

On constate donc que τ peut se voir comme un cas particulier de $\mathcal{V}$ [16].
On peut également aller un peu plus loin et profiter du pouvoir de modélisation d'attaquant offert par les $\Pr(p)$. Pour ce faire, on pondère chaque faute f par des poids w_f tels que :

— $\Pr(p_f) = w_f$
— $\sum_{f \in \mathcal{F}} w_f = 1$

On obtient alors le taux de vulnérabilité suivant :

$$\mathcal{V} = \sum_{f \in \mathcal{F}_S} w_f \tag{4.9}$$

Les poids w_f peuvent être choisis arbitrairement pour refléter les choix de l'attaquant : par exemple, l'attaquant peut favoriser certaines fautes, auxquelles on associe un poids

16. On peut également retrouver trivialement N en calculant $\mid \mathcal{F} \mid \times \tau$.

relativement plus grand. Ceci permet de contrer le paradoxe de dilution, puisqu'il est possible de mettre à zéro le poids correspondant aux fautes qui n'ont aucune chance de succès (par exemple, les fautes qui correspondent aux NOP rajoutés dans le code).

4.4.5. Sensibilité locale

4.4.5.1. Métrique locale

Dans cette section, on compare les problématiques des métriques locales avec celles, discutées à la section 4.4.1, des métriques globales. Les métriques locales posent le problème du regroupement des fautes en fonction des paramètres. De même que pour les métriques globales, il est possible d'exprimer une métrique locale « naturelle » comme le nombre de fautes réussies avec certaines valeurs de paramètres de l'équipement d'attaque. Comme cette métrique est exprimée dans l'espace des paramètres de l'équipement, il est cependant difficile de faire le lien avec le code, et donc avec les contremesures.

Dans la littérature, Berthomé *et al.* [Ber+12] donnent une métrique basée sur la proportion d'attaques réussies dans l'outil en fonction de l'adresse d'injection de faute. On peut voir cette métrique comme une application de la métrique globale τ paramétrée par les différentes adresses mémoires.

Aujourd'hui il n'est pas aisé de comparer au niveau local plusieurs implémentations, car la correspondance d'adresses/emplacements entre plusieurs binaires différents n'est pas directe. Par contre, le paradoxe de l'augmentation de la surface d'attaque ne se pose pas : ajouter du code ajoute de nouvelles adresses qui n'ont pas de correspondance dans le binaire d'origine, les vulnérabilités les concernant ne peuvent donc pas se comparer à la version d'origine. De même, le paradoxe de la dilution ne se pose pas : l'ajout de NOP constitue de nouvelles instructions qui n'ont pas d'impact sur les valeurs de la métrique locale pour les instructions déjà existantes. En revanche, on a toujours le même problème de traitement équivalent de fautes de probabilités différentes. Enfin, exprimer une métrique locale dans le cas de fautes multiples indépendantes est un problème ouvert, dont on discutera au chapitre 5.

4.4.5.2. Construction du taux de sensibilité

La sensibilité locale $\mathcal{S}_\ell$ est la métrique locale que nous proposons. Pour calculer un point de cette métrique pour un emplacement ℓ, on calcule la métrique $\mathcal{V}$ suivant la formule de l'équation (4.3), mais en remplaçant $\mathcal{F}_S$ par l'ensemble $\mathcal{F}_S^\ell$ des fautes conduisant à une attaque réussie et ayant pour caractéristique spatiale ℓ.

Pour un emplacement ℓ on a donc :

$$\mathcal{S}_\ell = \sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S^\ell} \Pr(F = f \mid p) \cdot \Pr(p) \tag{4.10}$$

$\mathcal{S}_\ell$ transpose l'avantage de $\mathcal{V}$ sur τ et N aux métriques locales : la métrique tient compte de la probabilité d'occurrence de chaque type de faute.

Exemple 4.8
La figure 4.4 compare les métriques locales $\mathcal{S}_\ell$ et τ_ℓ (définie comme le rapport du nombre d'attaques réussies à une adresse donnée sur le nombre d'attaques total effectué à cette adresse) sur la plage d'adresses [`0x100030`, `0x10006c`] sur une implémentation `VerifyPIN` sur carte A sous injection électromagnétique (l'implémentation et la plage d'adresses sont différentes de notre exemple fil rouge).

Les différences principales entre les deux métriques locales sont la présence ou l'absence de pondérations des fautes suivant leur probabilité d'occurrence. Ces différences peuvent amener à des différences d'interprétation des différents « points sensibles » du code. Sur l'exemple, on observe ainsi qu'à l'adresse `0x100042`, la métrique τ_ℓ donne une forte sensibilité, tandis que $\mathcal{S}_\ell = 0$. En effet, l'analyse des vulnérabilités montre que les fautes à cette adresse sont des fautes qui nécessitent une perturbation $\text{read}_m(EEPROM, 0, b)$ (pour b quelconque non nul), ce qui est exclu de notre modèle de faute probabiliste pour la carte A.

À l'inverse, on observe aux adresses `0x100064` et `0x100066` que $\mathcal{S}_\ell$ est élevée tandis que que τ_ℓ est faible. En effet, les fautes à ces adresses reposent sur la perturbation $\text{read}_m(EEPROM, a, 0)$ (pour a quelconque non nul), qui est un événement particulièrement probable dans notre modèle de faute probabiliste.

Il est difficile de conclure sur la validité de la comparaison présentée sur cet exemple, car il faudrait pouvoir comparer avec la métrique locale « naturelle » comme référence. Malheureusement, cette métrique s'exprime dans l'espace des paramètres d'équipement, et la conversion vers l'espace des paramètres du code n'est pas triviale.

4.5. Langage de spécification

Dans cette section, on présente GISL [17], le langage de spécification d'architecture de CELTIC. GISL permet aux utilisateurs de l'outil d'adapter le simulateur au composant sécurisé à simuler : c'est par GISL et un fichier de spécification que les fonctions de décodage et d'exécution d'instructions sont spécialisées suivant la machine à simuler. GISL est donc un élément essentiel de CELTIC. Comme souvent avec ce type de langages, GISL désigne à la fois le langage en lui-même et son implémentation. Dans la suite, on ne fera donc pas de distinction entre ce qui relève de l'un ou de l'autre. On commence par rappeler les objectifs spécifiques de GISL au sein de CELTIC, en détaillant les points difficiles. Pour le problème de l'encodage et du décodage, on présente l'approche spécifique de GISL, en comparant avec les solutions à l'état de l'art.

4.5.1. Motivations de GISL

Les langages de spécification d'architecture permettent de décrire l'architecture de systèmes informatiques. On les distingue des langages de description de matériel de type VHDL ou Verilog en ce qu'ils décrivent l'architecture d'un point de vue fonctionnel, et non son implémentation. Comme pour le reste de CELTIC (voir section 4.2.1.1),

17. *Generic IC Specification Language*, ou « Langage générique de spécification de composants ».

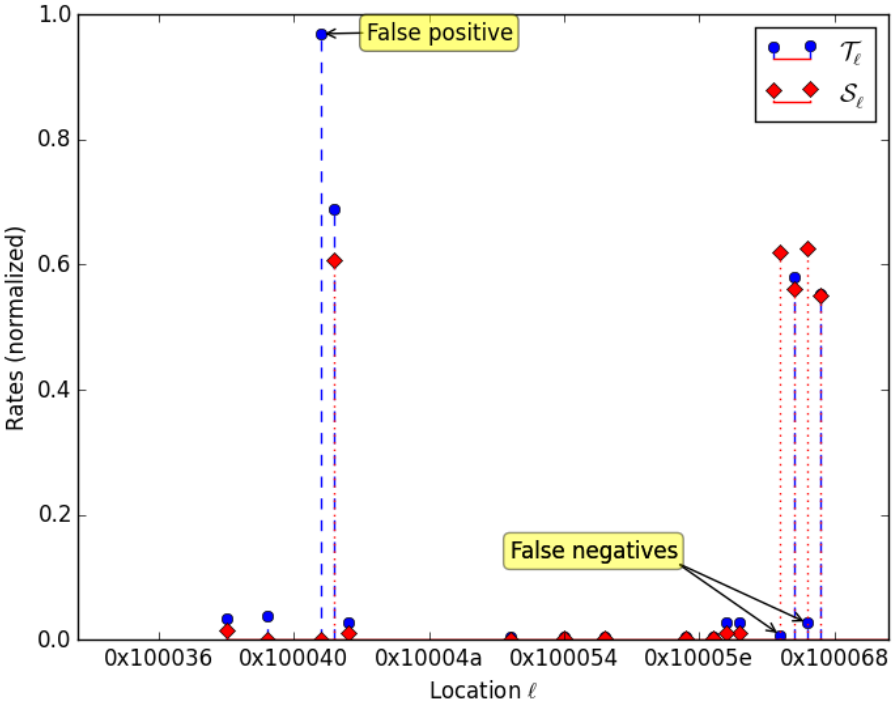


FIGURE 4.4. – Comparaison de τ_ℓ avec $\mathcal{S}_\ell$

l'objectif de GISL est de donner un bon niveau de contrôle (expressivité) sur la spécification aux évaluateurs tout en garantissant des performances décentes en termes de rapidité du simulateur produit, ce qu'on obtient en limitant la précision de modélisation du composant. Ainsi, GISL se positionne en tant que langage de description de jeux d'instruction à un niveau d'abstraction élevé se contentant de décrire le minimum pour permettre la simulation d'un processeur avec le niveau de précision d'une instruction. Les objectifs de GISL sont : (1) permettre la spécification de composants variés, (2) être facile à utiliser pour les évaluateurs et (3) permettre la génération de simulateurs efficaces. Plus précisément :

1. Variété de composants. On peut classer approximativement les jeux d'instructions en deux grandes familles : les architectures *Complex Instruction Set Computer* (CISC) et *Reduced Instruction Set Computer* (RISC). Les architectures CISC se distinguent par un grand nombre d'instructions aux sémantiques proches, opérant avec des modes d'adressages différents. Par contraste, les architectures RISC proposent peu d'instructions qui n'opèrent que sur des registres, ainsi qu'un nombre réduit d'instructions pour communiquer avec la mémoire (on parle aussi de jeux d'instructions *load/store*). Spécifier des architectures CISC ou RISC facilement (pour le spécifieur) implique donc que le langage soit adapté à ces deux types d'architectures.

2. Facilité d'utilisation. GISL vise des temps de spécification courts pour des utilisateurs qui sont les évaluateurs CESTI, c'est-à-dire des utilisateurs qui ont des connaissances en architecture matérielle, mais sans être nécessairement experts dans le domaine. Le point qui nécessite le plus de travail est la spécification de l'encodage d'une instruction. Pour les jeux d'instructions très simples, le nombre d'instructions peut être suffisamment réduit pour être intégralement spécifié manuellement, mais dans la plupart des jeux d'instructions actuels, le nombre de mode d'adressage (pour les CISC) ou l'encodage des paramètres (pour les RISC) interdit cette approche. Avec GISL, on vise donc un langage qui permet de spécifier la sémantique du composant ciblé instruction par instruction, simplement en lisant la documentation du jeu d'instructions, mais en limitant au maximum les répétitions dans la spécification et en factorisant la sémantique commune à plusieurs instructions d'encodages différents.

3. Génération de simulateurs efficaces. Les simulateurs générés à partir de langages de spécification ont un coût supplémentaire par rapport aux simulateurs écrits manuellement. En particulier, la phase de décodage des instructions est souvent moins efficace. Les simulateurs manuels s'appuient sur la connaissance des différents formats d'instructions pour décoder l'instruction en reconnaissant son format à partir de son code opération. Une difficulté pour les simulateurs générés à partir de langages de spécification est donc de générer une fonction de décodage efficace sans connaître les formats d'instruction.

4.5.2. Caractéristiques de GISL

Dans cette section, on présente rapidement une vue d'ensemble de GISL, puis on précise la solution adoptée pour la spécification de l'encodage d'une instruction (sec-

tion 4.5.2.1) et l'algorithme de décodage d'une instruction (section 4.5.2.2), qui sont deux points critiques. GISL effectue l'analyse syntaxique d'un fichier d'entrée et en extrait un arbre abstrait qui est ensuite utilisé pour vérifier la correction du fichier d'entrée[18] et pour la simulation. Un fichier d'entrée est constitué de plusieurs sections, qui décrivent :

- Les propriétés de base du composant à simuler : taille de la mémoire, largeur du registre PC, nombre et taille des registres, taille des mots d'instruction, *endianness* des données et du code.
- Les zones mémoires et leurs permissions d'écriture/lecture/exécution (améliore la précision de la simulation en détectant les opérations illégales sur la mémoire)
- D'éventuelles correspondances entre des noms et des indices de registres (pour les architectures qui ont des registres nommés).
- La liste des instructions du jeu d'instructions, avec pour chaque instruction son nom, ses différents encodages et sa sémantique. Cette sémantique est exprimée dans une syntaxe classique proche de celle du `C` enrichie de quelques opérateurs spécialisés pour la manipulation de données (concaténation de données, accès indicés aux bits d'une donnée, opérateur d'accès mémoire).

4.5.2.1. Encodage d'une instruction

GISL permet la spécification de composants avec une précision au niveau de l'instruction, et l'encodage d'une instruction constitue un des défis principaux du langage.

Dans un jeu d'instructions, l'encodage d'une instruction définit la ou les manière(s) de représenter l'instruction sous la forme d'une suite de bits. Les programmes, constitués de séquences d'instructions, sont ainsi codés sous la forme d'un binaire en utilisant l'encodage des instructions. CELTIC accepte en entrée de tels programmes binaires, et doit les décoder pour pouvoir en extraire la séquence d'instructions. GISL permet donc d'exprimer le lien entre l'encodage binaire et la sémantique d'une instruction. La difficulté dans GISL a été de trouver une syntaxe équilibrée en termes d'expressivité et de simplicité.

Encodage naïf. Une solution naïve peut être de spécifier un encodage, puis la sémantique. Le code 4.8 relie ainsi l'encodage #8 0x42, #8 0x21 (c'est-à-dire la suite de bits 0b0100001000100001) à la sémantique de l'instruction `ADD` d'addition entre le registre A et le registre Y[19]. L'avantage de cette solution est sa simplicité. Son inconvénient est qu'il est nécessaire d'associer à chaque encodage sa sémantique. Or, les composants de type CISC définissent de nombreuses instructions à la sémantique proche, opérant avec des paramètres différents. Ainsi, l'encodage #8 0x40, #8 0x21 peut désigner une addition entre le registre X et et registre Y. Dans l'encodage naïf, cela se traduit par le codes 4.9, qui est presque identique au code 4.8. Cette répétition augmente le temps de saisie et augmente le risque d'erreur de spécification.

18. Les éventuelles erreurs de syntaxe et de sémantique (typage des expressions, existence des identifiants utilisés) sont reportées.

19. Les encodages et les registres choisis ont valeur d'illustration : ils ne correspondent pas à un jeu d'instructions existant.

</>Code GISL

```
ADD
#8 0x42, #8 0x21;
{
  result = reg_A + reg_Y;
  reg_A = result;
  /* flags */
  if (result == 0) {
    flag_Z = 1;
  } else {
    flag_Z = 0;
  }
}
```

Listing 4.8. – Encodage naïf #1

</>Code GISL

```
ADD
#8 0x40, #8 0x21;
{
  result = reg_X + reg_Y;
  reg_X = result;
  /* flags */
  if (result == 0) {
    flag_Z = 1;
  } else {
    flag_Z = 0;
  }
}
```

Listing 4.9. – Encodage naïf #2

Table d'encodage pour les CISC. Pour résoudre ce premier problème, GISL propose le support de tables d'encodage. Cette solution est inspirée des *templates* `C++`, qui permettent d'écrire du code paramétré, dont les instances seront générées à la compilation, et qui sont notamment utilisés comme support des conteneurs à types génériques dans la bibliothèque standard. Parmi les langage de spécification d'achitecture, ArchC [Rig+04], qui exploite la plateforme SystemC pour générer des simulateurs performants et précis, utilise une syntaxe proche de `C++` [20] et notamment emploie le concept de *templates* de code `C++` afin de réutiliser les modes d'adressages déclarés entre différentes instructions.

20. SystemC est une bibliothèque `C++`

Exemple 4.9 *Tables d'encodage.*
Dans les tables d'encodage proposées par GISL, la sémantique ne manipule plus directement les registres et autres paramètres, mais des variables `data[0]`, `data[1]`, ..., `data[n]` génériques. À chaque sémantique manipulant ces variables génériques correspondent plusieurs encodages, qui partagent la sémantique tout en spécialisant la valeur des variables génériques avec les paramètres effectifs. Ainsi, la répétition de la sémantique d'addition des codes 4.8, 4.9 est évitée dans le code 4.10. Dans ce code, les deux encodages ont la même sémantique, sauf que le premier remplace `data[0]` par le registre A et le second par le registre X.

Par rapport à ArchC, la solution de GISL ne repose pas sur `C++`, mais sur un langage dédié. La complexité d'utilisation est donc beaucoup moins élevée (en particulier, la syntaxe des *templates* `C++` est connue pour être difficile à appréhender).

</>Code GISL

```
ADD
#8 0x42, #8 0x21 : reg_A, reg_Y;
#8 0x40, #8 0x21 : reg_X, reg_Y;
{
  result = data[0] + data[1];
  data[1] = result;
  /* flags */
  if (result == 0) {
    flag_Z = 1;
  } else {
    flag_Z = 0;
  }
}
```

Listing 4.10. – Table d'encodage

Motifs d'encodage pour les RISCs. Les tables d'encodage se comportent mal avec les composants de type RISC, qui ont des jeux d'instructions qu'on pourrait qualifier de « densément codés ». Dans les CISCs, l'encodage correspondant à une sémantique et ses paramètres est généralement arbitraire (ou représentatifs de choix d'architectures invisibles au niveau du jeu d'instructions), mais leur nombre reste faible. Dans les RISCs, ce n'est plus le cas : les paramètres sont encodés de façon régulière et documentés dans la spécification. En contre-partie, ils sont beaucoup plus nombreux : l'instruction Thumb-2 (ARM) d'addition peut être encodée par 7 bits ayant pour valeur 0xc, suivi de 3 bits désignant l'indice du registre opérande gauche de l'addition, 3 bits désignant l'indice du registre opérande droit, et 3 bit pour l'indice du registre destination. Si ces différents encodages doivent être spécifiés selon la méthode des tables d'encodage, l'évaluateur doit alors spécifier $2^{3+3+3} = 512$ encodages différents (mais tous très similaires) pour la seule instruction d'addition. Pour remédier à ce second problème, on propose dans GISL une syntaxe de reconnaissance par motifs (*pattern matching*), qui permet d'exprimer simplement la relation fonctionnelle entre l'encodage et les paramètres de

l'instruction. Dans GISL, les motifs sont soit des noms de variables libres, soit des conditions booléennes faisant intervenir une variable libre, et ils sont assortis d'une taille en nombre de bits. Par exemple, le motif : `#3 m` reconnaît n'importe quelle suite de 3 bits, tandis que `#4 (a < 0xd && a >= 9)` ne reconnaît que les suites de 4 bits inférieures à `0xd` et supérieures ou égales à 9 (soit les valeurs `0b1001`, `0b1010`, `0b1011`, `0b1100`). Parmi les langages de spécification d'architecture, GDSL [SKS12] propose déjà d'utiliser la reconnaissance de motifs. Cependant la solution retenue dans GISL est assez différente de celle proposée par GDSL, qui exprime la sémantique des instructions sous une forme rappelant le standard ML [Mil+97], utilise ses motifs uniquement sous forme de suites de bits (apparemment sans possibilité d'exprimer directement des conditions booléennes), et ne propose pas de tables d'encodage pour partager la sémantique (elle est partagée via des appels de fonctions).

Exemple 4.10 *Utilisation des motifs.*
En utilisant les motifs dans GISL, l'encodage d'addition Thumb-2 précédent peut s'écrire : `#7 0xc, #3 m, #3 n, #3 d`. En GISL, les motifs nommés dans l'encodage peuvent être utilisés dans les paramètres. Les motifs qui apparaissent dans tous les encodages attachés à une même sémantique peuvent également apparaître directement dans la sémantique. Le code 4.11 donne une sémantique simplifiée de l'instruction d'addition en Thumb-2.

Les tables d'encodages et les motifs peuvent se combiner dans GISL (plusieurs encodages avec paramètres pour une sémantique), ce qui permet à GISL de s'adapter à des jeux d'instructions RISC, CISC, ou « hybrides » entre ces deux paradigmes.

</>Code GISL

```
ADD
#7 0xc, #3 m, #3 n, #3 d : d, n;
#8 0x44, #1 DN, #4 (m != 0xd), #3 dn : DN.dn, DN.dn;
{
  result = reg[data[1]] + reg[m];
  reg[data[0]] = result;
  /* flags */
  if (result == 0) {
    flag_Z = 1;
  } else {
    flag_Z = 0;
  }
}
```

Listing 4.11. – Motifs d'encodage

4.5.2.2. Algorithme de décodage

L'algorithme de décodage permet au simulateur de reconnaître l'instruction correspondant à une suite précise de bits, et fournit la fonction `decode()` utilisée par l'algo-

rithme de simulation présenté à la section 4.2.1.2. Il est important que cet algorithme soit efficace en temps, car la phase de décodage compte parmi les plus longues de l'algorithme de simulation.

Une solution naïve de décodage consiste à tester chaque motif défini dans le fichier de spécification jusqu'à ce que la suite de bits soit reconnue. Cette solution présente l'inconvénient d'avoir une complexité linéaire en le nombre de motifs définis, et de ne pas permettre la détection des motifs ambigus. Deux motifs sont ambigus lorsqu'il existe au moins une suite de bits reconnue par les deux motifs. Comme un même encodage ne peut pas correspondre à plusieurs instructions différentes, l'ambiguïté signifie qu'il existe une erreur dans le fichier de spécification. Dans la méthode naïve, ces erreurs ne sont pas détectées (sauf à tester exhaustivement la reconnaissance par motifs).

Une solution classique plus efficace consiste à utiliser un arbre de recherche, c'est-à-dire effectuer des séquences de comparaisons afin de diviser l'espace des motifs candidats à chaque comparaison. Dans CELTIC, l'arbre de recherche est construit automatiquement en fonction des différents motifs déclarés dans le fichier de spécification. À notre connaissance, GISL est le seul langage de spécification d'architecture où la reconstruction de l'arbre de décodage est automatique (c'est-à-dire, qui ne nécessite pas d'être spécifiée par l'utilisateur), mais où le temps de décodage est néanmoins logarithmique en le nombre de motifs reconnaissables. Par exemple, dans [Res+03], les auteurs parlent d'une complexité de décodage en $\mathcal{O}(n \log(n))$ pour n motifs dans le langage de spécification (similaire à un tri). GDSL ne précise pas la complexité du décodage par ses motifs.

L'algorithme de GISL fonctionne en deux étapes, s'effectuant une fois durant la lecture du fichier de spécification.

1. Mise des motifs d'encodage sous forme canonique.
2. Construction de l'arbre de recherche à partir de la forme canonique des motifs d'encodage.

Lors de la première étape, les différents motifs sont transformés en vecteurs de bits, chaque bit valant soit 0, soit 1, soit la valeur _, qui indique que 1 et 0 sont tous les deux reconnus par le motif. Par exemple, le vecteur 1__0 reconnaît les suites 1000, 1010, 1100 et 1110. Pour un motif de longueur k, jusqu'à k vecteurs sont ainsi générés. Par exemple, le motif `#4 (m != 0xd)` génère les vecteurs :

$$\begin{cases} 0 & \_ & \_ & \_ \\ 1 & 0 & \_ & \_ \\ 1 & 1 & 1 & \_ \\ 1 & 1 & 0 & 0 \end{cases}$$

La génération des vecteurs a une complexité temporelle linéaire en le nombre de motifs déclarés par l'utilisateur.

L'idée de la seconde étape est de découper l'espace des candidats possibles suivant la valeur d'un bit de la suite à reconnaître. Il faut le faire de façon à exclure le maximum de candidats en observant le minimum de bits, il faut donc choisir de manière optimale la séquence d'indices à lire dans la suite de bits. Pour une collection de m vecteurs de longueur k nommés $B_0, B_1, \ldots, B_{m-1}$, on calcule pour tout indice i entre 0 et $m-1$ les quantités $n_0(i)$ et $n_1(i)$, c'est-à-dire le nombre de vecteurs parmi les

$B_0, B_1, \ldots, B_{m-1}$ qui contiennent respectivement un 0 ou un 1 à l'indice i. Les valeurs _ ne sont pas comptées. L'indice i_S d'une comparaison optimale pour partitionner l'espace des $B_0, B_1, \ldots, B_{m-1}$ est donné par la formule :

$$i_S = max_{i \in \{0 \ldots (m-1)\}}(min(n_0(i), n_1(i))).$$

En effet cette formule donne l'indice où la comparaison découpe l'espace en deux parties de cardinal les plus proches possibles. L'algorithme est ensuite appliqué récursivement pour les sous-espaces ainsi construits. L'exemple 4.11 montre la construction de l'arbre de décodage.

Exemple 4.11 *Construction de l'arbre de décodage.*

B_0	=	0	1	_	_	_	1	_	_	
B_1	=	1	0	_	_	_	0	_	_	
B_2	=	1	1	_	_	_	0	_	_	$\Longrightarrow \{B_0, B_3\}, \{B_1, B_2\}$
B_3	=	1	_	0	_	_	1	_	_	
min	=	1	1	0	0	0	2	0	0	

Dans la première étape de cet exemple, on partage l'espace des $\{B_0, B_1, B_2, B_3\}$ en deux sous-espaces $\{B_1, B_2\}$ et $\{B_0, B_3\}$, puis, on applique l'algorithme récursivement sur chacun de ces deux sous-espaces pour obtenir l'arbre final donné par la figure 4.5. Une fois l'arbre construit, il suffit d'effectuer la recherche dans l'arbre pour savoir par quel motif une suite de bits est reconnue. Pour reconnaître par exemple la suite 11011010, on regarde le $6^{ème}$ bit : c'est un 0, on regarde donc ensuite le $2^{ème}$ bit : c'est un 1. Le dernier candidat est donc B_2. Enfin, il faut vérifier que la suite 11011010 est entièrement reconnue par B_2. Cette étape est nécessaire car la suite 01011010 conduit à B_2 dans l'arbre, mais n'est en fait reconnue par aucun motif. Le coût du décodage (parcours de l'arbre et reconnaissance par le motif candidat) reste inférieur à un décodage naïf (tentatives exhaustives de reconnaissance par tous les motifs).

Par ailleurs, cet algorithme de construction permet de garantir la détection des motifs ambigus. En effet un sous-espace de motifs ambigus donne $S = 0$ dans la formule plus haut. Ceci permet donc de détecter ces problèmes et de renvoyer une erreur sans coût supplémentaire.

4.6. Conclusion

Dans ce chapitre, on a présenté CELTIC, un outil pensé pour l'évaluateur. Ses principales contributions sont son caractère paramétrable, qui lui permet de prendre en charge de nombreux composants grâce à un langage de spécification léger d'utilisation, son support des modèles de faute probabiliste, et ses facultés de traitement des résultats à l'aide de métriques globales et locales qui, pour la première fois dans le domaine des attaques par perturbation, tiennent compte d'un modèle d'attaquant pour calculer le résultat.

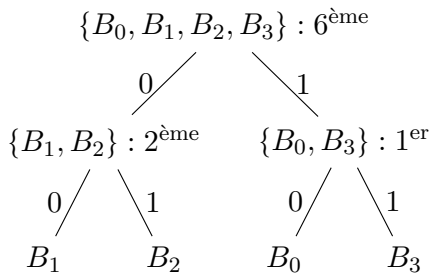


FIGURE 4.5. – Arbre résultant de l'algorithme de partitionnement.

4.6.1. C'est au pied du mur...

CELTIC a été testé sur quatre composants différents : deux puces Cortex M, dont l'une sécurisée, et deux cartes avec des jeux d'instructions propriétaires (CISC). Sur ces composants, CELTIC a simulé et évalué diverses implémentations de fonctions : le fameux exemple `VerifyPIN` bien sûr, mais également la commande `GetChallenge` qui effectue la génération sécurisée d'un challenge aléatoire dans un tampon, les fonctions cryptographiques de chiffrement symétrique `DESEncrypt` et `AESEncrypt` et asymétrique `RSA-CRT`, et une fonction de vérification de primalité d'un entier `IsPrime`.

CELTIC a été utilisé pour valider des résultats dans une évaluation réelle, et pour comprendre des résultats obtenus au laser dans une autre évaluation pour laquelle la revue de code manuelle n'était pas assez précise pour arriver à une explication.

En termes de durée d'utilisation, il faut à un évaluateur environ cinq jours pour ajouter un nouveau jeu d'instructions à l'aide de GISL [21], et, suivant la complexité de l'exemple, entre dix minutes et deux jours pour configurer l'évaluation d'une nouvelle application. L'essentiel de ce temps est passé pour initialiser le contexte d'appel de la fonction, déterminer les adresses de début et de fin de simulation, et interpréter pourquoi certaines attaques réussissent.

4.6.2. Intégration du taux de vulnérabilité dans le processus d'évaluation

Comment intégrer le taux de vulnérabilité $\mathcal{V}$ qu'on a présenté dans ce chapitre au processus d'évaluation ? On a vu que cette métrique permet de prédire le facteur *elapsed time* sans effectuer de tests de pénétration.

Cependant, pour disposer de ce pouvoir prédictif, $\mathcal{V}$ repose sur la méthode d'inférence de modèle de faute vue au chapitre 3, qui doit avoir été appliquée pour la plate-forme matérielle sous-jacente à l'application évaluée. On a vu au chapitre 1 que, dans le cadre des cartes à puce, un produit est typiquement soumis à deux évaluations (voir figure 1.5 page 14) : une première pour la plate-forme matérielle (la carte), puis une seconde pour l'application. Ces deux processus sont typiquement réalisés par deux CESTI différents. Pour bénéficier du $\mathcal{V}$ durant l'évaluation d'une application, l'idéal serait donc que l'inférence de modèle de faute soit effectuée par le CESTI qui évalue la plate-forme

21. Cette opération n'est nécessaire que pour les évaluations de composants qui utilisent un jeu d'instructions non encore spécifié.

matérielle, et transmise au CESTI en charge de l'évaluation logicielle, par exemple via le rapport technique d'évaluation.

Par ailleurs, il existe déjà aujourd'hui des normes de cotation d'attaque (voir le tableau 1.3 page 17) que les évaluateurs sont tenus de respecter. En particulier, les tests de pénétration sont obligatoires dans le processus et ne peuvent pas être remplacés par l'utilisation de CELTIC et le taux $\mathcal{V}$. Néanmoins, cette métrique peut aider à préparer et guider les attaques physiques. L'apport de modèles d'attaquant offre également une réflexion plus fine sur les capacités dévolues aux attaquants dans l'attaque.

4.6.3. Et les fautes multiples, dans tout ça ?

Tout au long de ce chapitre, on n'a pas évoqué le problème des fautes multiples. Celles-ci ont le potentiel de démultiplier le pouvoir de l'attaquant, puisqu'il peut par exemple cibler la fonctionnalité et désamorcer les contre-mesures durant une même exécution.

CELTIC supporte les fautes multiples, dans le sens vu à la section 2.2.4, c'est-à-dire que les algorithmes présentés s'étendent naturellement au cas des fautes multiples indépendantes, au prix d'un nombre de fautes possibles à tester exponentiel en le nombre de fautes indépendantes considérés. Dans la pratique, cette extension naïve est bien trop lente pour tester des exemples réels. Par ailleurs, les fautes multiples indépendantes complexifient également la question du traitement des résultats. Ces différentes problématiques sont traitées dans le chapitre 5, qui détaille les réflexions et les expérimentations conduites sur le sujet au cours de la thèse.

Chapitre 5

Fautes multiples indépendantes

Si $2 + 2 = 5$, alors je suis le Pape.

(Attribué à Bertrand Russel)

Résumé du chapitre

Dans ce chapitre, on présente les travaux effectués sur le problème des fautes multiples indépendantes, où un attaquant peut effectuer plusieurs perturbations de la carte durant une seule exécution de son code. Ce problème implique une grande combinatoire qui complexifie considérablement la simulation de l'injection de fautes ainsi que le traitement des résultats. Pour pouvoir traiter ce problème particulier, on commence par proposer une formalisation du problème de l'injection de faute en utilisant des graphes d'exécution classiques. Cette formalisation nous permet de proposer plusieurs méthodes de réduction du problème de l'injection de fautes en les modélisant sur les graphes d'exécution. On s'intéresse ensuite au traitement des résultats dans le contexte des fautes multiples, et en particulier au problème des « attaques redondantes », c'est-à-dire les attaques multiples où toutes les fautes ne sont pas nécessaires pour que l'attaque réussisse. On propose ensuite une extension au cas des fautes multiples de la métrique de traitement globale des résultats à l'aide de modèles d'attaquant particuliers. Enfin, on présente un moyen d'améliorer grandement les performances de CELTIC en appliquant les techniques de compilation à la volée à l'injection de fautes.

Sommaire

5.1. Introduction

On parle de fautes multiples *indépendantes* pour désigner les fautes générées par plusieurs perturbations différentes au cours d'une même exécution, par opposition à des fautes touchant plusieurs instructions ou emplacement mémoires (par exemple, remplacer quatre instructions consécutives par des NOPs) et causées par une unique perturbation. Par abus de langage, on pourra toutefois utiliser seulement le terme « fautes multiples » pour désigner les fautes multiples indépendantes. Ce chapitre n'aborde pas les fautes causées par une unique perturbation. Depuis leur début dans la littérature en 2007, les attaques par injection de fautes multiples indépendantes augmentent considérablement les pouvoirs de l'attaquant. On verra à la section 5.5.2.3 un exemple d'une telle attaque mise concrètement en œuvre au CESTI. Comme on va cependant le voir, les attaques par fautes multiples complexifient grandement l'analyse de robustesse à l'injection de fautes, que cette analyse soit manuelle ou effectuée à l'aide d'un outil. L'objectif de ce chapitre est de bien poser les différents problèmes causés par l'injection de fautes multiples, et de présenter quelques pistes prometteuses pour faciliter l'analyse de robustesse dans ce cas de figure.

Dans ce chapitre, on fait l'hypothèse qu'une attaque par fautes multiples est constituée de k fautes choisies dans un même modèle de faute sous-jacent comprenant n fautes différentes possibles.

5.1.1. Fautes multiples et explosion combinatoire

Pour comprendre l'explosion combinatoire du problème des fautes multiples, on propose d'évaluer le nombre de fautes multiples possible en choisissant k fautes parmi les n fautes différentes possibles du modèle sous-jacent. De plus, pour calculer des temps effectifs d'exécution de la campagne de tests exhaustifs, on considère avec optimisme qu'il est possible d'effectuer 100 000 simulations fautées par seconde.

Dans ces conditions, il existe k parmi n fautes distinctes dans le modèle miroir $\mathcal{M}_{\text{mirror},k}$, c'est-à-dire C_n^k, où C_n^k désigne les coefficients binomiaux :

$$C_n^k = \frac{n!}{k!(n-k)!}$$

La complexité algorithmique du nombre de possibilités à considérer dépend donc des valeurs respectives de k et n. On considère trois cas : l'exemple 5.1 où $k = \frac{n}{2}$, l'exemple 5.2 où $k \ll n$, et l'exemple 5.3, où l'on souhaite calculer toutes les combinaisons pour toutes les valeurs de k entre 0 et n.

Exemple 5.1 *Cas où $k = \frac{n}{2}$.*
Dans ce cas, le comportement asymptotique de C_n^k est équivalent au premier ordre à :

$$C_n^{\frac{n}{2}} \sim \frac{2^n}{\sqrt{\frac{n}{2}\pi}} \text{[BE14]}$$

Même pour un « petit » modèle de faute NOP, tester exhaustivement un programme de 100 instructions (100 fautes possibles) donne 1.00×10^{29} combinaisons de 50 fautes possibles différentes, soit 3.20×10^{16} années de calcul.

Exemple 5.2 *Cas où $k \ll n$.*
Dans ce cas, on a un modèle de faute « large » (NOP sur un long programme, remplacement d'instruction, etc.) et on souhaite passer au multi-fautes pour de petites valeurs de k (tester avec 2, 3 ou 4 fautes). On a alors l'équivalence au premier ordre :

$$C_n^k \sim \frac{n^k}{k!} \text{[BE14]}$$

Pour donner une idée, pour un « gros » modèle de faute de 10 000 fautes possibles, il existe 5×10^7 possibilités (8 minutes de calcul) en faute double, 2×10^{11} possibilités (19.2 jours de calcul) en faute triple et 4.16×10^{14} possibilités (132 années de calcul) en faute quadruple.

Exemple 5.3 *Calcul pour toutes les valeurs de k dans $\{0, \ldots, n\}$.*
Enfin, on considère également la situation où l'on souhaite calculer toutes les possibilités de fautes simples et multiples pour k dans l'intervalle $\{0, \ldots, n\}$. Dans ce cas, le nombre de possibilités est :

$$\sum_{k=0}^{n} C_n^k = 2^n$$

Inutile de rappeler au lecteur que de tels nombres de fautes sont impossibles à traiter exhaustivement (pour 100 fautes, il faut calculer pendant un temps approximativement égal à 2.6×10^7 **fois l'âge de l'univers**).

La conclusion de ces trois exemples est qu'il devient très rapidement impossible d'utiliser une stratégie d'exécution dynamique exhaustive des fautes pour résoudre le problème de l'injection de fautes multiples avec les ordinateurs actuels. Cependant, ce n'est pas le seul problème. Même en supposant qu'on sache maîtriser la combinatoire pour tester toutes les fautes possibles, le traitement des résultats produits serait également très complexe : la compréhension d'attaques qui exploitent plusieurs fautes est plus difficile, et le nombre de résultats bien supérieur.

5.1.2. Solutions proposées

Tout au long de ce chapitre, on propose plusieurs solutions pour chercher à maîtriser la combinatoire de la génération des résultats, ainsi que celle de leur traitement. Pour la génération des résultats, on propose trois approches essentiellement complémentaires : (1) utiliser des heuristiques pour résoudre des sous-cas du problème général, (2) diminuer la complexité avec une réduction du problème, et (3) améliorer les performances de simulation avec un compilateur à la volée. Pour le traitement des résultats, on part du problème des attaques dites « redondantes », qui concernent les attaques multiples dans lesquelles certaines fautes sont superflues. On se sert de cet exemple pour bien définir l'objectif, et donner des pistes de résolution. On montre également comment on peut étendre les métriques et les modèles d'attaquants vus au chapitre 4 (voir section 4.4).

5.2. Analyse guidée par les contre-mesures

L'objectif initial des attaques par fautes multiples est de compromettre les propriétés de sécurité des composants sécurisés même en présence de contre-mesures [KQ07]. Un scénario courant d'attaque par fautes multiples est donc de faire une partie des fautes pour compromettre la sécurité, et le reste pour défaire les contre-mesures. Pour analyser la robustesse d'une application contre l'injection de fautes multiples indépendantes, on peut donc effectuer deux sous-analyses : une analyse de robustesse des contre-mesures, et une analyse de robustesse de l'exemple une fois les contre-mesures défaites. L'exemple 5.4 donne un exemple de fautes détectées par cette analyse.

Cette méthode réduit le nombre de cas à traiter en permettant de considérer les fautes multiples comme un ensemble de fautes simples à objectifs différents. La complexité résultante varie en fonction du nombre de fautes permettant de déjouer les contre-mesures. En notant p_i le nombre de fautes permettant de déjouer la contre-mesure i, dans une attaque constituée de k fautes visant à déjouer $k-1$ contre-mesures, si $\forall i \in \{1,\ldots,k-1\}$, p_i est très petit devant le nombre n de fautes possibles dans le modèle, alors le nombre de possibilités est de l'ordre de :

$$\left(\prod_{i\in\{1,\ldots,k-1\}} p_i\right) \times n$$

c'est-à-dire à nouveau linéaire en n.

Cette méthode est heuristique, car elle ne traite pas exhaustivement le problème des fautes multiples : en effet, on peut construire des fautes qui individuellement n'attaquent pas les contre-mesures, mais qui collectivement permettent de compromettre la sécurité,

comme le montre l'exemple 5.5. Cette méthode est pourtant proche de celle utilisée lors des revues manuelles : en présence de contre-mesures, les évaluateurs vérifient s'il est possible de la déjouer avec une faute, puis d'attaquer directement une propriété de sécurité avec une autre. La section 5.3.2.3 présente une systématisation de cette heuristique qui permet de traiter tous les cas.

Exemple 5.4 *Exemple de faute détectée par l'heuristique*
On considère l'implémentation de `VerifyPIN` du chapitre 4 (voir listing 4.1 page 100), en particulier la boucle :

</>Code source

```
11  for(i = 0; i < size; i++) {
12    if(a1[i] != a2[i]) {
13      diff = BOOL_TRUE;
14    }
15  }
16  if (i != size) {
17    countermeasure();
18  }
```

L'analyse guidée par les contre-mesures peut proposer par exemple une première faute pour ne pas exécuter la boucle (par exemple, l'inversion de la comparaison `i < size`) ligne 11, et une deuxième faute pour éviter la contre-mesure ligne 16 (par exemple, en inversant la condition `i != size`).

Exemple 5.5 *Exemple de cas non traité par l'heuristique*
On considère l'implémentation donnée à l'exemple 5.4, et les fautes qui inversent la comparaison `if(a1[i] != a2[i])` (ligne 12) à chaque tour de boucle (pour `i` $\in \{0, \ldots, \texttt{size}-1\}$). Individuellement, une seule de ces fautes ne permet pas de forcer l'authentification, ni de déjouer la contre-mesure de la ligne 16 si une autre faute fait sortir de la boucle plus tôt. Pourtant, collectivement, ces fautes permettent de forcer l'authentification, en forçant toutes les comparaisons des chiffres du PIN. Ces fautes ne sont pas considérées par l'heuristique présentée, bien qu'elles constituent un danger évident pour la sécurité de l'application.

5.3. Réduction du problème de l'injection multiple

Dans cette section, on présente des algorithmes qui visent à maîtriser la combinatoire associée aux injections de fautes. Contrairement à la section 5.2, on traite cette fois du cas général et non de sous-cas. L'intuition des algorithmes de réduction est d'identifier des « classes » d'exécutions aux résultats similaires, ce qui permet ensuite d'obtenir le résultat pour toute la classe en le calculant sur un représentant de la classe. Pour

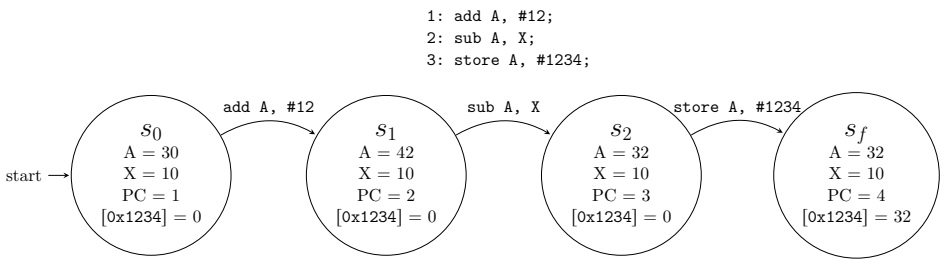


FIGURE 5.1. – Trace d'exécution représentée par un graphe d'exécution

faciliter les raisonnements, on propose une formalisation du problème de l'injection de faute qui repose sur un graphe d'exécution. On peut présenter les méthodes de réduction plus facilement sur ce graphe.

5.3.1. Formalisation du problème

Pour cette formalisation du problème de l'injection de faute, on commence par décrire les exécutions par un processeur d'un programme comme un graphe d'exécution, puis on définit le problème de l'injection de faute dans ce cadre, avant de l'étendre au cas des fautes multiples.

5.3.1.1. Graphe d'exécution

Le choix de notre formalisme part des traces d'exécution produites par CELTIC. On utilise un graphe dont les nœuds sont les états du composant qui fait des exécutions du programme. Un même graphe peut alors représenter l'exécution nominale, ainsi que les divergences dans l'exécution causées par les fautes.

On fait l'hypothèse que le composant a un comportement déterministe et entièrement défini par une fonction `exec`, qui associe à un état s son successeur `exec`(s) après l'application d'une « étape » de calcul du composant [1]. Dans ces conditions, on peut voir une exécution nominale (en l'absence de faute) comme un état initial s_0, un état final s_f, et une suite d'états intermédiaires entre s_0 et s_f, chaque état étant obtenu à partir du précédent par application de la fonction `exec`. On peut voir un chemin sur ce graphe comme une représentation de la trace d'exécution (voir figure 5.1). Dans ce graphe, chaque nœud a au plus une seule transition sortante : celle qui correspond à l'application de la fonction `exec`. Si un nœud a plus d'une transition entrante, alors l'exécution est une boucle infinie et ne termine pas (voir figure 5.2).

Précisément, un nœud contient les données suivantes :

— Les valeurs contenues dans les différents emplacements mémoire (flash, ram, registres) du composant.

— La liste (potentiellement infinie) des entrées utilisateurs qui auront lieu durant l'exécution du programme [2].

1. On peut modéliser la notion d'« étape » avec la granularité souhaitée : opération atomique dans le composant, cycle d'horloge, exécution d'instruction, etc.

2. Si on veut pouvoir gérer des générateurs de hasard, on peut inclure leurs sorties dans la liste des entrées. On conserve ainsi le déterminisme de `exec`.

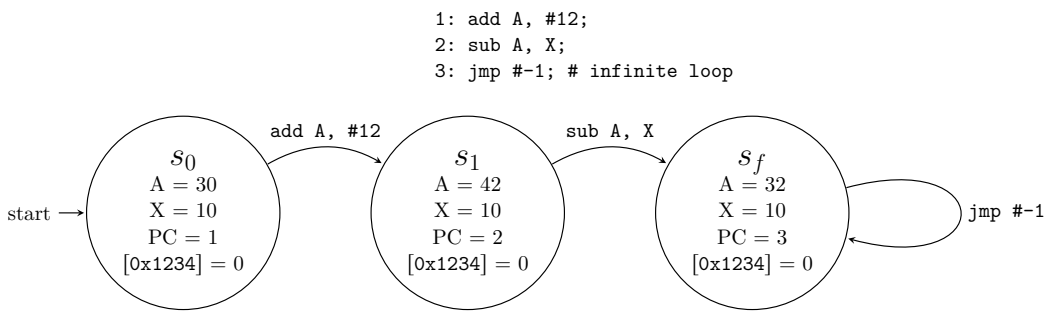


FIGURE 5.2. – Graphe d'exécution avec boucle infinie

— L'état interne du processeur, qui contient les informations nécessaires pour rendre `exec` déterministe et entièrement définie. Ces données comprennent par exemple la partie lue dans la mémoire d'une instruction avant son décodage final, une éventuelle *pipeline* d'instructions, la prochaine opération à effectuer (décodage, troisième *statement* dans la sémantique d'une instruction, etc.).

L'idée d'utiliser les chemins sur un graphe pour représenter des traces d'exécution est relativement classique. On en trouve au moins un exemple en 2010 par Porto *et al.* pour de l'optimisation dynamique [Por+10]. Plus proche de nous, c'est aussi une représentation choisie par Goubet *et al.* pour construire les prédicats de chemin envoyés à un SMT-solver dans leur outil [Gou15], qu'on a déjà décrit au chapitre 2. La variante qu'on propose aujourd'hui a la particularité que, si l'on atteint un nœud connu (même état), et en l'absence de faute supplémentaire, un seul chemin est possible par la suite (cette propriété sera utile à la section 5.3.2.1). On dira que le graphe est « replié ».

Cette représentation dispose d'une granularité fine, ce qui a l'avantage de donner une bonne expressivité, et l'inconvénient de comporter beaucoup de nœuds à considérer. Il faut en effet considérer des chemins et des nœuds différents si l'état initial est modifié (par exemple les entrées utilisateurs).

5.3.1.2. Problème de l'injection de fautes

Dans le cadre défini à la section 5.3.1.1, on précise le problème de l'injection de fautes, d'abord dans le cas général, puis dans le cas le plus intéressant en pratique.

Comme on a pu le voir tout au long de ce manuscrit, l'injection de faute peut se comprendre comme une perturbation du fonctionnement nominal du composant. Dans notre formalisme, ceci revient à introduire une transition arbitraire d'un nœud (état fauté) vers un autre (état résultant de la faute). La figure 5.3 montre l'application d'une faute permettant d'atteindre le nœud s'_1 au lieu du nœud s_1 à partir du nœud s_0.

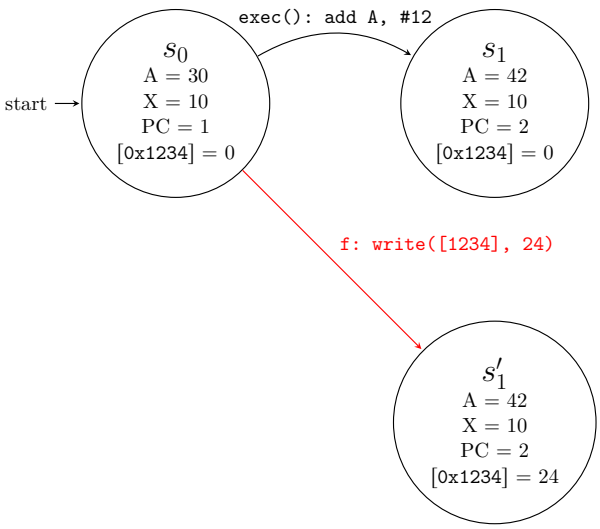


FIGURE 5.3. – Graphe d'exécution fauté

Dans ces conditions, on donne une définition du problème de l'injection de fautes :

Définition 5.1 *Problème de l'injection de fautes*
Soient un ensemble d'états $S_{\mathcal{N}} = \{s_0, s_1, \ldots, s_n\}$ tels que :

$$\forall i \in \{0, \ldots, n-1\},\ s_{i+1} = \texttt{exec}(s_i)$$

et un ensemble d'états $S_{\mathcal{S}}$ représentant les états qui compromettent la sécurité de l'application (par exemple, qui vérifient un oracle), et que l'attaquant cherche à atteindre. Le problème de l'injection de faute $\text{FIP}(S_{\mathcal{N}}, S_{\mathcal{S}})$ est la question de l'existence d'une ou plusieurs fautes (transitions arbitraires) permettant d'atteindre un élément de $S_{\mathcal{S}}$ depuis un élément de $S_{\mathcal{N}}$.

Posé de cette manière, le problème de l'injection de faute est facile à résoudre. Comme rien ne contraint la nature des fautes, on se retrouve avec deux cas triviaux :
— $S_{\mathcal{S}}$ ou $S_{\mathcal{N}}$ sont vides, alors $\text{FIP}(S_{\mathcal{N}}, S_{\mathcal{S}})$ n'a pas de solution.
— $S_{\mathcal{S}}$ et $S_{\mathcal{N}}$ sont non-vides, alors n'importe quelle transition d'un élément de $S_{\mathcal{N}}$ vers un élément de $S_{\mathcal{S}}$ est une solution.
Ce résultat revient à constater qu'un attaquant omnipotent peut trivialement arriver à ses fins. On va donc contraindre les fautes (transitions) acceptées pour modéliser un attaquant moins puissant et plus réaliste.

5.3.1.3. Contraint par un modèle de faute

Dans le problème de l'injection de fautes contraint par un modèle de faute (MBFIP, pour *Model-Bounded Fault Injection Problem*), les transitions fautées ne sont plus arbitraires, mais font partie d'un modèle de faute spécifié. En section 3.2.3 du chapitre 3,

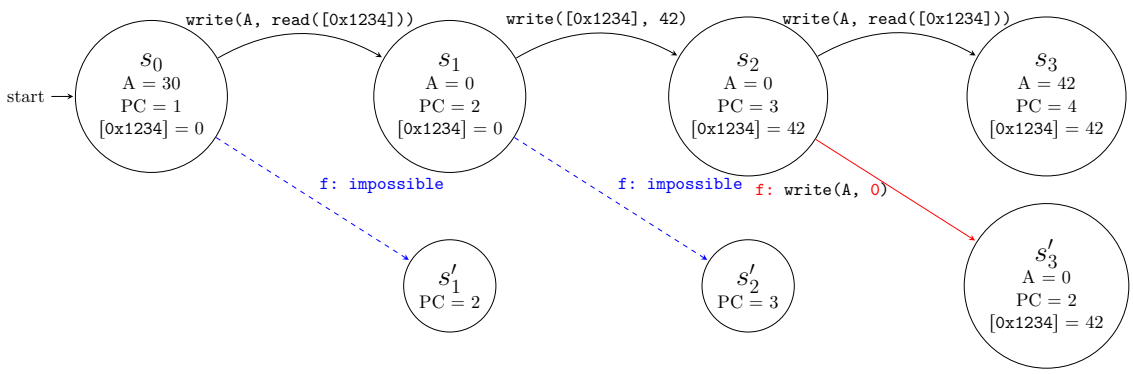


FIGURE 5.4. – Fautes contraintes par le modèle de faute

on définit un modèle de faute probabiliste comme un ensemble de générateurs de fautes assortis chacun d'une probabilité d'occurrence [3]. Une transition fautée f sortante du nœud s est donc possible seulement si f est une opération prévue par le modèle de faute, et s vérifie les prédicats prévus par le générateur pour cette opération. On définit le problème de l'injection de faute contraint par un modèle de faute :

Définition 5.2 *Problème de l'injection de faute contraint par un modèle de faute.* Soient un ensemble d'états $S_{\mathcal{N}} = \{s_0, s_1, \ldots, s_n\}$ tels que :

$$\forall i \in \{0, \ldots, n-1\}, s_{i+1} = \texttt{exec}(s_i)$$

et un ensemble d'états $S_{\mathcal{S}}$ représentant les états qui compromettent la sécurité de l'application que l'attaquant cherche à atteindre, et un modèle de faute $\mathcal{M}$. Le problème de l'injection de faute contraint par le modèle $\mathcal{M}$, MBFIP$(S_{\mathcal{N}}, S_{\mathcal{S}}, \mathcal{M})$, est la question de l'existence d'une ou plusieurs fautes (transitions arbitraires) du modèle $\mathcal{M}$ permettant d'atteindre un élément de $S_{\mathcal{S}}$ depuis un élément de $S_{\mathcal{N}}$.

La figure 5.4 montre un exemple de ce qui est possible ou non avec un modèle de faute $\mathcal{M}_0$, qui permet de fauter une lecture d'un emplacement EEPROM de valeur a non nulle vers 0. Ainsi, on ne peut pas fauter l'état s_1 car sa prochaine opération n'est pas une opération de lecture mais une opération d'écriture, et on ne peut pas fauter l'état s_0 car son opération de lecture renvoie déjà la valeur 0. On peut en revanche fauter l'état s_2.

Méthode de résolution. Pour résoudre un MBFIP, on propose la méthode de la *campagne exhaustive*, qui consiste à tester toutes les transitions fautées appartenant au modèle considéré sur l'ensemble d'états $S_{\mathcal{N}}$. Si une faute conduit à un état de $S_{\mathcal{S}}$, alors le MBFIP a une solution.

3. L'aspect probabiliste n'intervient pas dans le problème de l'injection de fautes contraint par un modèle de faute.

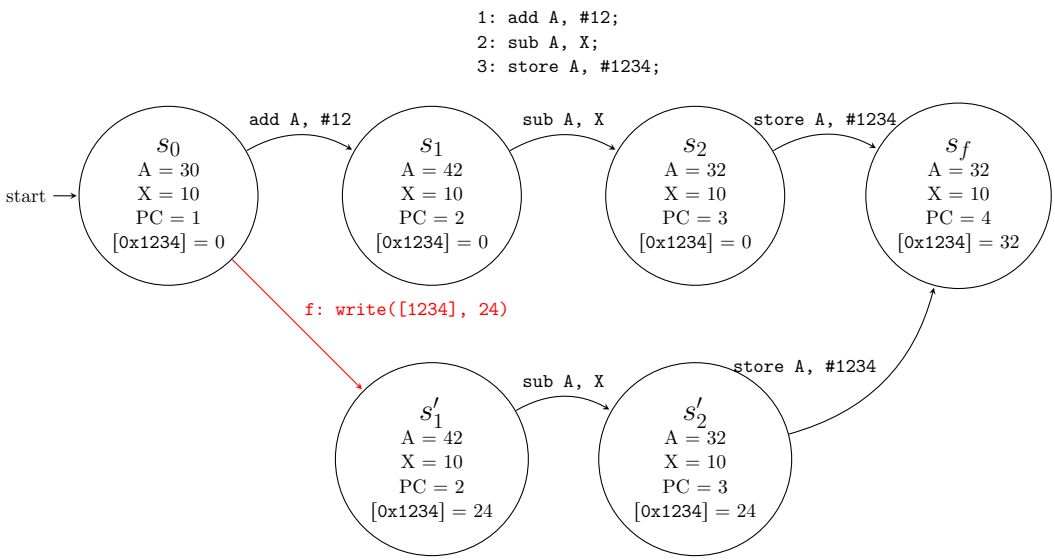


FIGURE 5.5. – Graphe avec deux chemins semblables

Parallèle avec le fonctionnement de CELTIC. On a détaillé au chapitre 4 le fonctionnement de l'outil CELTIC. Celui-ci effectue une première simulation non fautée de référence, puis des exécutions fautées. On peut voir la première simulation non fautée comme la construction de l'ensemble $S_{\mathcal{N}}$ des états nominaux, et les simulations fautées comme une exploration du graphe des états atteignables à partir de transitions fautées. Les états $S_{\mathcal{S}}$ sont déduits de l'oracle de victoire (voir section 4.2.3). Les attaques réussies trouvées par CELTIC sont les solutions du problème.

5.3.2. Techniques de réduction

Munis de la formalisation du graphe d'exécution, on présente deux techniques de réduction du problème de l'injection de faute : la réduction des états, qui vise à rassembler des états équivalents en un seul nœud du graphe, et la réduction des transitions, qui vise à rassembler des fautes équivalentes en une seule transition.

5.3.2.1. Réduction des états

On propose la propriété suivante : si on arrive à un certain état s, et qu'on ne fait plus de fautes (c'est-à-dire qu'on n'utilise plus que la fonction `exec`), on connaît l'état final s_f, et il ne dépend pas des états qui précèdent s. En particulier, si la dernière faute autorisée par le modèle nous amène dans un état connu, alors on a déjà exploré les états successeurs de cet état, et par conséquent on sait déjà si l'attaque est une solution du problème ou non.

Observons le graphe de la figure 5.5. On constate que les états s_1 et s_2 sont identiques respectivement aux états s'_1 et s'_2, à la valeur de la mémoire à l'adresse 0x1234 près, qui n'intervient pas dans le calcul (elle est affectée à `0x32` avant d'être lue). La méthode de réduction des états consiste à considérer de tels états, qui ne diffèrent que par des valeurs qui ne sont pas utilisées dans l'exécution, comme identiques. On dira que ces

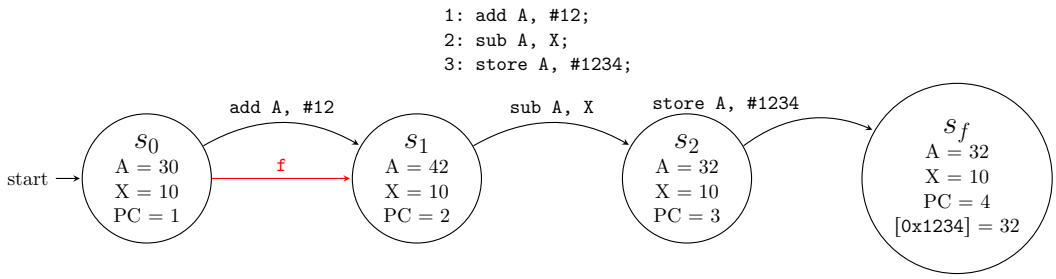


FIGURE 5.6. – Graphe après réduction des chemins

états sont équivalents. Ainsi, si la dernière faute d'une exécution conduit dans un état équivalent à un état connu, on connaît l'état final et donc on sait si l'attaque est une solution du problème ou non.

Cette méthode permet donc de réduire la taille de l'espace à explorer, en regroupant les états équivalents et en arrêtant l'exécution dès qu'on atteint un état équivalent à un état connu. La figure 5.6 montre le graphe obtenu par la méthode de réduction. Après l'exécution nominale qui amène à construire s_1, s_2 et s_3, on peut arrêter l'exécution fautée dès qu'on atteint s_1, et ne pas recalculer s_2 et s_3 qui sont déjà connus. Cette méthode devient intéressante quand l'exécution est très coûteuse, par exemple pour l'évaluation de long programmes. L'inconvénient de cette méthode est qu'il faut savoir déterminer quand deux états sont équivalents, ce qui n'est pas trivial : une bonne base peut être de rendre équivalents tous les états qui ne diffèrent que par des valeurs qui ne sont pas utilisées dans l'exécution nominale. Toutefois, c'est approximatif, car des valeurs non utilisées dans l'exécution nominale peuvent se révéler utiles dans les chemins fautés (obligeant à séparer des états auparavant considérés comme équivalents), et des valeurs utilisées dans l'exécution nominale pourraient être peu sensibles aux variations (par exemple, le code PIN entré par l'utilisateur a le même comportement tant qu'il est différent du code PIN de référence).

5.3.2.2. Réduction des transitions

De même que pour les états, on définit deux fautes comme équivalentes si elles conduisent au même état. La technique de réduction des transitions vise donc à diminuer le nombre de transitions à suivre en traitant les fautes équivalentes comme une unique transition. La technique du *register pruning* est une des principales techniques de réduction du problème de l'injection de fautes utilisées dans le domaine de la tolérance aux fautes. Dans le contexte d'un modèle de faute consistant en une faute qui modifie la valeur contenue dans un registre, le *register pruning* considère comme équivalente toutes les fautes comprises entre un point de définition (*def*) du registre et sa prochaine utilisation (*use*). On parle de chaîne *def/use*, et on considère donc comme équivalentes toutes les fautes sur une même chaîne *def/use*. La figure 5.7 donne une illustration de ce principe avec deux graphes équivalents (séparés par le symbole ≡). Le *register pruning* est tellement important pour le domaine que Schirmeier *et al.* rapportent dans [SCS15] qu'il a été « redécouvert » de très nombreuses fois dans la littérature (les auteurs citent Smith *et al.* [Smi+95] et Güthoff et Sieh [GS95] comme les premiers à expliquer le

concept avec concision, puis Benso *et al.* [Ben+98], Barbosa *et al.* [Bar+05] ou encore Hari *et al.* [Har+12]) [4].

Implémentation.
En termes d'expérimentations, on a implémenté un prototype d'extension de la technique de *register pruning* à un modèle de faute quelconque. Pour ce faire, on observe les emplacements lus et écrits par chaque faute du modèle, et on a considéré comme équivalentes toutes les fautes écrivant la même valeur aux mêmes emplacements le long de la même chaîne *def/use*. Cependant, le résultat s'est révélé décevant dans ces conditions : on a constaté une augmentation du temps d'exécution de 150%. Cette « optimisation négative » peut s'expliquer par plusieurs facteurs :

— La qualité de l'implémentation du prototype, qui reposait beaucoup sur `ruby`, beaucoup plus lent que le cœur `C++` du simulateur.
— Le modèle de faute utilisé, qui comportait assez peu de fautes équivalentes suivant ce critère.
— Les programmes testés qui étaient relativements courts, et pour lesquels il était donc plus rapide de faire la simulation en entier plutôt que de chercher les chaînes *use/def* pour raccourcir la simulation.

Avec des programmes plus longs, et surtout dans le cadre des fautes multiples, il serait toutefois intéressant de reprendre cette implémentation pour observer son impact sur les performances.

Une autre piste pour découvrir les fautes équivalentes serait de tirer parti de certaines correspondances entre les fautes de plusieurs modèles de faute : par exemple, on établit dans [Dur+16] que remplacer des instructions de comparaison ou de branchement par des NOP dans le modèle *InstSkip* est équivalent dans certaines conditions à inverser un test dans le modèle *BranchChoice*. Dans ce cas, il est possible de traiter un NOP sur une comparaison comme une faute équivalente à un NOP sur le branchement qui suit. La figure 5.8 montre une réduction de transitions qui exploite cette correspondance. La technique de réduction des transitions est particulièrement utile en fautes multiples, puisqu'elle réduit directement le nombre de fautes du modèle à traiter, ce qui a un fort impact sur le nombre de fautes multiples possibles (voir section 5.1.1 pour les calculs de combinatoire). La section 5.3.2.3 présente d'ailleurs une technique de réduction spécialement adaptée aux fautes multiples.

5.3.2.3. Réduction des transitions et fautes multiples

L'objectif de cette section est d'étendre la méthode heuristique présentée à la section 5.2, qui proposait de répartir les fautes entre celles destinées à éliminer les contremesures et celles pour attaquer la propriété de sécurité. Pour mettre en œuvre cette

4. On note qu'il est également possible d'utiliser le *register pruning* avec une réduction des états. Par ailleurs, la réduction des transitions implique souvent une réduction des états. Une différence majeure entre les deux techniques de réduction est que la réduction des états raccourcit des simulations, tandis que la réduction des transitions diminue leur nombre.

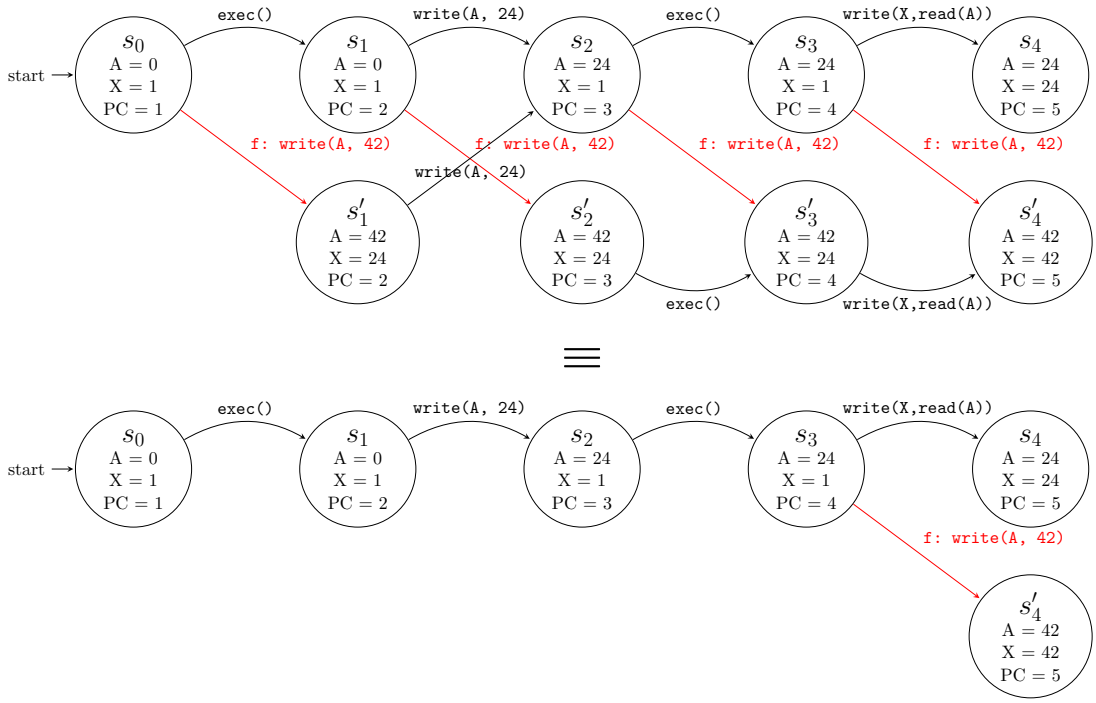


FIGURE 5.7. – Réduction des transitions par *register pruning*

extension, par exemple sur les fautes doubles, il faut commencer par explorer tout le graphe pour les fautes simples. Ensuite, on attribue à chaque nœud le nombre minimal de fautes nécessaires pour pouvoir atteindre le nœud : 0 pour un nœud atteignable directement par `exec()` sans faute, 1 pour un nœud atteignable exclusivement avec une faute. La technique de réduction consiste ensuite à considérer les fautes doubles dont la première faute conduit à un nœud numéroté 0 comme équivalentes à simplement la deuxième faute de la faute double. En effet, la deuxième faute ne peut conduire dans ces conditions qu'à un nœud numéroté 1, qu'on a déjà exploré dans l'analyse en faute simple. Il suffit donc de considérer les fautes doubles dont la première faute conduit à un nœud numéroté 1. On peut encore réduire davantage l'espace en considérant toutes les fautes simples qui conduisent au même nœud numéroté 1 comme équivalentes. Dans ces conditions, explorer en faute double est équivalent à considérer les fautes simples qui partent d'un nœud numéroté 1. La méthode s'étend naturellement pour les fautes triples et les autres fautes multiples. Cette méthode permet de traiter l'exemple 5.5 présenté à la section 5.2 : les fautes individuelles permettent d'accéder à des états du graphe inaccessible par un simple `exec()`, et sont donc prises en compte par la méthode. En revanche, il n'est pas clair que retenir tous les états atteignables de simulation pour les numéroter soit réalisable en pratique.

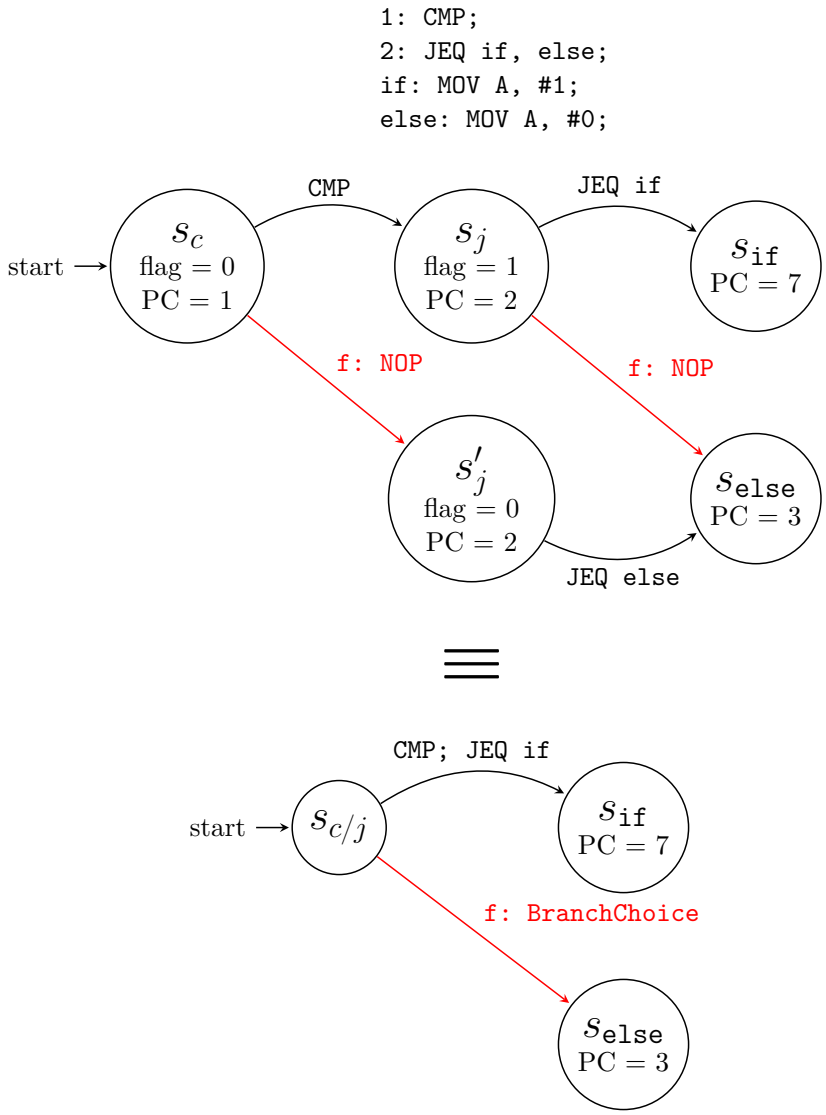


FIGURE 5.8. – Réduction des transitions par correspondance de modèles

5.4. Amélioration de performances avec un compilateur à la volée

Dans cette section, on présente une technique pour faciliter le passage aux fautes multiples d'un outil d'analyse dynamique tel que CELTIC. La technique est complètement orthogonale aux méthodes de réduction de la complexité abordées dans le reste de ce chapitre. Il s'agit d'utiliser une technique d'optimisation avancée, la compilation à la volée, pour accélérer la simulation du code à évaluer. La particularité dans CELTIC est qu'on effectue de l'injection de faute, donc la compilation à la volée doit s'effectuer d'une façon qui tienne compte de la modification par les fautes d'une partie de la sémantique du programme.

5.4.1. Motivation

L'algorithme de simulation de CELTIC a été présenté à la section 4.2.1.2 du chapitre 4, et repose sur l'interprétation de la sémantique des instructions qui composent le code à simuler, instructions elles-mêmes décodées depuis les octets du binaire. Cette opération de décodage est lente pour les jeux d'instructions modernes (tels que ARM), dont les instructions disposent d'opérandes encodées de façon complexe [Bra+04]. De plus, l'utilisation d'un générateur de simulateur tel que GISL rend d'autant plus lente l'interprétation des instructions [Wag+13]. Dans le cas de GISL, la sémantique d'une instruction est obtenue à l'exécution par le parcours d'un arbre abstrait. Ceci se traduit par de multiple appels de fonction du simulateur pour exécuter les *statements* qui constituent la sémantique d'une instruction, et pour évaluer les expressions à l'intérieur de chaque *statement*. Ainsi, dans le jeu d'instructions ARMv7-M, l'instruction ADDI (ajout d'une constante numérique à un registre) est constituée de 56 *statements* et d'environ 225 expressions, pour un total d'environ 300 appels de fonction par le simulateur pour l'exécution de la sémantique. Si on reprend l'exemple 4.7 du chapitre 4 d'une campagne d'injection de fautes exhaustive avec un modèle de faute « complexe » sur un code `VerifyPIN`, on a effectué environ 80 000 attaques de 130 instructions chacune, soit un total d'environ $80\,000 \times 130 \times 300 = 3.12 \times 10^9$ appels de fonction par le simulateur pour l'exécution des sémantiques. Or, le temps passé à évaluer l'arbre abstrait dans chaque simulation n'est pas strictement nécessaire, car au moment de la simulation du programme, son code est connu. L'idée de la compilation à la volée, proposée dès les années 80 dans l'implémentation du langage SELF de Sun [CUL89], est donc de générer du code correspondant à la sémantique du programme simulé à destination de la plate-forme d'exécution du simulateur (x86-64). Cette compilation est dite « à la volée » (ou *Just In Time* en Anglais), car elle s'effectue au dernier moment avant l'exécution du code généré. Les gains proviennent d'une part de la possibilité d'effectuer des optimisations natives pour la plate-forme de destination, d'autre part de la possibilité de mettre en cache le code généré (qui est beaucoup plus rapide que l'interprétation), pour le réutiliser tel quel (sans nouvelle compilation) lorsque c'est nécessaire. Dans beaucoup de cas, ces gains contre-balancent largement le coût induit par la compilation. C'est la solution retenue dans de nombreuses machines virtuelles comme celles pour Java [Cra+97], Microsoft .Net [KS01], ou encore Lua [Pal05]. Dans le cas de l'injection de faute, on peut espérer un fort gain : une simulation avec injection de faute réutilise

l'intégralité du code généré, à quelques légères modifications près (qui ciblent quelques instructions du code à évaluer). Si on reprend l'exemple 4.7, rien qu'en nombre d'appels de fonction, on peut espérer faire correspondre à des blocs de 5 instructions du binaire d'origine un seul appel de fonction, soit un total de $80\,000 \times \frac{130}{5} = 2.08 \times 10^6$, soit plus de 1000 fois moins d'appels [5].

5.4.2. Algorithme de compilation/simulation à la volée

Dans le cadre de notre simulateur pour l'injection de faute, l'algorithme de génération est divisé en deux étapes :

1. **Reconstruction d'une partie du graphe de flot de contrôle** [6] : Une première simulation a lieu pour construire une partie du graphe de flot de contrôle, c'est-à-dire identifier des groupes d'instructions qui s'exécutent en séquence. Cette étape permet d'associer un bloc de base à chaque instruction rencontrée durant la simulation. La reconstruction du graphe de flot de contrôle n'est pas complète, on ne s'intéresse qu'aux instructions rencontrées dans les chemins de code effectués par la simulation choisie.
2. **Génération du code pour les blocs de base du graphe de flot de contrôle** : Dans cette étape, on génère du code natif x86-64 pour chaque bloc de base du graphe de flot de contrôle construit à l'étape précédente. Plus précisément, chaque bloc se voit associé une fonction x86-64 qui prend en entrée un état de simulation, et qui renvoie cet état modifié par la sémantique des instructions (du programme d'origine) qui composent le bloc.

Une fois que le code a été généré, l'algorithme de simulation présenté au listing 5.1, reprend essentiellement celui de la section 4.2.1.2. La différence est qu'on cherche si l'instruction correspondante à l'adresse pointée par le registre `PC` est au début d'un bloc pour lequel on a généré du code. Si c'est le cas, on exécute la fonction x86-64 générée à partir du bloc, afin de mettre à jour l'état de simulation. À défaut, c'est l'interprétation qui est utilisée, jusqu'à retrouver une instruction de début de bloc.

Lors de la campagne d'injection, on fait une simulation comme au listing 5.1, mais en marquant le bloc contenant l'instruction où la faute se produit temporellement comme « invalide ». Les instructions de ce bloc sont alors interprétées, ce qui permet d'appliquer la sémantique de la faute injectée. Ceci signifie qu'il faut bien choisir la taille (nombre d'instructions) des blocs d'instructions : des blocs plus longs permettent une meilleure optimisation et donc une simulation plus rapide, mais augmentent le nombre d'instructions à interpréter en cas de faute.

5. Bien que les appels de fonctions aient un coût (en particulier les appels de fonction virtuels comme ceux de l'évaluation d'expression), il est petit en général face au coût d'exécution de la fonction. Ce calcul donne donc une estimation du gain espéré, mais on s'attend à bien moins impressionnant dans la pratique.

6. On parle bien ici de graphe de flot de contrôle, et plus de graphe d'exécution comme dans la section 5.3.

</>Code source

```
def simulateJIT(simu, initial_state, iter_max, end_addr):
    iter = 0
    state = initial_state.copy
    trace = []
    while (state.pc != end_addr and
        not state.hasError and
        iter < iter_max):
            if simu.isAtStartOfBlock(state):
                # current inst is start of block; execute it
                block = simu.getBlock(state)
                iter = iter + block.size()
                simu.exec(state, block)
            else:
                # unrecognized block, default to interpretation
                iter = iter + 1
                inst = simu.fetch(state, trace)
                simu.exec(state, inst, trace)
    return trace, state
```

Listing 5.1. – Algorithme de simulation

Implémentation.
L'implémentation d'un compilateur à la volée est complexe. Un premier facteur de cette complexité est que le domaine de la compilation est complexe de façon générale (optimisations depuis le langage source, génération de code natif, optimisation de ce code natif). Un autre facteur est que la compilation à la volée vise à concilier deux objectifs opposés : générer un code très efficace, et le générer rapidement. Pour s'abstraire de ces difficultés, on a utilisé une bibliothèque tierce pour la génération de code natif : le *framework* LLVM [LA04]. LLVM est une infrastructure constituée d'une représentation intermédiaire (LLVM IR, pour *Intermediate Representation*), d'une machine virtuelle, d'un générateur de code natif pour de très nombreuses plate-formes, et de passes d'analyse et d'optimisation pour la représentation intermédiaire. À la différence des autres écosystèmes de compilation, LLVM a été pensé dès l'origine autour de sa représentation intermédiaire et de la possibilité d'enchaîner plusieurs passes d'optimisation sur ce format. LLVM est utilisé dans de nombreux projets [a], dont le compilateur `clang`, « concurrent » de `gcc`. Les possibilités de compilation à la volée sont exploitées notamment dans `Mono`, l'implémentation libre de la machine virtuelle .Net de Microsoft à destination de Linux, Android et iOS [b]. CELTIC génère donc des instructions au format intermédiaire LLVM à partir de l'arbre abstrait construit par GISL. Une fois le LLVM IR généré, on utilise directement le générateur de code de LLVM pour obtenir du x86-64.

Une première implémentation basée sur la bibliothèque `libjit` [c] a été réalisée, mais cette dernière n'offre qu'une documentation très minimale, n'est plus maintenue, et s'est révélée plutôt instable, précipitant le changement vers LLVM.

a. http://llvm.org/ProjectsWithLLVM/
b. Voir http://www.mono-project.com/docs/advanced/runtime/docs/llvm-backend/.
c. https://savannah.gnu.org/git/?group=libjit

5.4.3. Résultats

On a fait quelques tests préliminaires de performances de notre compilateur à la volée. Plus précisément, on a testé une implémentation ARMv7-M de l'exemple `VerifyPIN` provenant de FISSC. On observe que la simulation sans faute est environ **34 fois plus rapide** en *JIT* qu'en interprétation, avec 4 millions d'instructions ARMv7-M exécutées par seconde, soit environ 1000 instructions x86 pour émuler une instruction ARMv7-M.

En injection de faute, le gain est moins important, la campagne d'injection de faute sur un modèle de faute de remplacement d'instruction de 35 190 attaques est environ **11 fois plus rapide** en *JIT* qu'en interprétation, soit 5865 attaques effectuées par seconde.

Il semble possible d'améliorer encore les gains de performance avec plusieurs optimisations :

— La partie du graphe de flot de contrôle reconstruite ne concerne que l'exécution nominale. En particulier, les modifications du flot de contrôle causées par les fautes conduisent systématiquement à revenir à l'interprétation.
— L'algorithme de détection des instructions de fin de bloc est un peu naïf, et peut considérer certaines instructions comme fin de bloc à tort. Ceci réduit la taille des

blocs et en augmente le nombre. De meilleures optimisations seraient possibles sur des blocs plus longs.
— Il serait intéressant de compiler à la fois des blocs courts et des blocs longs, pour pouvoir profiter de la rapidité des blocs longs en l'absence de faute, et avoir à interpréter le moins possible d'instructions en utilisant les blocs courts en cas de faute.
— Pour les fautes multiples, il est envisageable de compiler à la volée des exécutions fautées par une faute simple pour diminuer la quantité d'instructions à interpréter.

5.5. Fautes multiples et analyse de robustesse

5.5.1. Le problème des attaques « redondantes »

Dans [Riv+14], Rivière *et al.* proposent la notion d'attaque « redondante ». Une attaque réussie est dite « redondante » quand elle fait intervenir toutes les fautes d'une autre attaque réussie. Par exemple, une attaque double constituée des fautes f_0 et f_1 qui réussit est redondante à l'attaque constituée de la faute f_0 si cette dernière réussit déjà. La notion de redondance exprime ainsi que f_1 « est inutile », puisque f_0 suffit à réussir l'attaque.

Les attaques redondantes posent problème pour le traitement des résultats : faut-il les inclure dans le résultat des analyses de robustesse ? Comme elles ne constituent pas de vulnérabilités supplémentaires, on peut penser que non. Pourtant, on peut imaginer un cas où des fautes multiples conduisent à des vulnérabilités, mais avec un risque évalué comme inférieur au seuil de cotation (voir section 1.1.3) si l'on exclut les attaques redondantes, et supérieur à ce seuil avec les attaques redondantes. Or, même si les attaques redondantes ne constituent pas de vulnérabilités supplémentaires, elles participent à augmenter la surface d'attaque pour les vulnérabilités existantes. Un attaquant qui fait des fautes doubles pourra exploiter les attaques même si elles sont redondantes ! Posée du point de vue des métriques, il semble difficile de répondre à la question de l'inclusion des attaques redondantes aux résultats de l'analyse. On propose donc de s'intéresser au point de vue de l'attaquant. Plus spécifiquement, on considère qu'une attaque réussie est dangereuse dès lors que l'attaquant choisit d'effectuer les fautes qui la constituent, et ce, que l'attaque soit redondante ou non.

La question de l'inclusion des attaques redondantes dans le résultat des analyses de robustesse est alors englobée par la question plus générale de déterminer un modèle d'attaquant pour les fautes multiples.

5.5.2. Modèles d'attaquant

À la section 4.4.3 du chapitre 4, on définit un modèle d'attaquant comme une façon de construire les lois de probabilités associées à $\Pr(p)$, qui est la probabilité que l'attaquant choisisse un certain jeu de paramètres p pour son équipement d'attaque dans l'ensemble $\mathcal{P}$ des valeurs de paramètres possibles. La notion provient directement de la construction de la métrique $\mathcal{V}$ qui vise à quantifier la sécurité globale d'une application contre l'injection de faute. Un avantage de la formalisation des modèles d'attaquant est son indépendance par rapport aux paramètres qui constituent $\mathcal{P}$. Il est donc facile

de l'étendre aux fautes multiples, en considérant les éléments de $\mathcal{P}$ comme l'union des paramètres de chaque perturbation successive (on applique déjà cette méthode à la section 3.2.3.2). On appelle $\mathcal{P}_k$ l'espace des paramètres d'équipement correspondant à k perturbations successives. Dans la suite, on présente plusieurs modèles utiles d'attaquant sur les espaces $\mathcal{P}_k$. Pour simplifier l'exposé, et sans perte de généralité, on considère le cas des fautes doubles, et donc de l'espace $\mathcal{P}_2$.

5.5.2.1. Attaquant à connaissance nulle

Le modèle d'attaquant à connaissance nulle en fautes multiples est similaire au même modèle en faute simple, présenté à la section 4.4.3.2. Il correspond à un attaquant qui n'a aucune connaissance de l'application ou de sa plate-forme et qui ne peut donc privilégier aucun paramètre de $\mathcal{P}_2$. On retrouve donc la formule du taux de vulnérabilité (voir définition à la section 4.4.2) :

$$\mathcal{V} = \frac{\sum\limits_{p \in \mathcal{P}_2} \sum\limits_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{\mid \mathcal{P}_2 \mid}$$

Dans la pratique, les attaquants commencent par des attaques en fautes simples, et il est très rare que cette première campagne d'injection ne leur apportent pas d'information sur l'application (voir section suivante).

5.5.2.2. Double faute guidée par les fautes simples

Dans la pratique, la campagne d'injection en faute simple permet souvent à l'attaquant de privilégier certaines fautes doubles. En effet l'attaquant remarque que certaines fautes simples aboutissent à des comportements différents du comportement nominal, comme une exécution plus courte, une écriture EEPROM ou une sortie différente. Ces fautes particulières servent alors de première faute lors des fautes doubles, ce qui permet de fixer l'une des deux fautes et ainsi de réduire drastiquement la taille de l'espace des paramètres à parcourir. Concrètement, si on note $p_0, \ldots, p_m$ les éléments de $\mathcal{P}_1$ qui correspondent à une faute simple aboutissant à un comportement non nominal, l'attaquant teste alors :

$$\mathcal{P}_{2,m} = \{(p_k, p) \mid k \in \{0, \ldots, m\} \wedge (p_k, p) \in \mathcal{P}_2\}$$

Avec un modèle uniforme sur cet espace, on trouve $\mathcal{V}$:

$$\mathcal{V} = \frac{\sum\limits_{p \in \mathcal{P}_{2,m}} \sum\limits_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{(m+1)\,(\mid \mathcal{P}_1 \mid -1)}$$

On note que le dénominateur est proportionnel à la taille de l'espace $\mathcal{P}_1$ et non plus à $\mathcal{P}_2$. Pour m suffisamment petit devant $\mathcal{P}_1$, un tel attaquant dispose donc de chances de succès beaucoup plus importantes qu'un attaquant à connaissance nulle. La méthode ressemble dans son principe à la méthode heuristique présentée à la section 5.2, sauf que l'attaquant n'a pas d'information directe sur les contre-mesures et s'appuie donc sur les différences de comportement. Dans ce modèle l'attaquant ne pourrait pas utiliser les fautes de l'exemple 5.5 page 145, car la faute simple sur la comparaison du PIN n'a pas d'effet visible sur l'exécution.

5.5.2.3. Cas particulier du DES/DES^{-1}

Pour certaines applications, l'attaquant peut avoir une connaissance préalable des contre-mesures mises en place, et des attaques à mettre en œuvre pour les défaire. C'est le cas de l'algorithme de chiffrement symétrique DES, pour lequel une contre-mesure courante est d'appliquer le déchiffrement DES^{-1} sur le chiffré pour vérifier qu'on retrouve bien le message clair d'origine [JT12 ; Kar+02 ; MSY06]. Si ce n'est pas le cas, ceci signifie que l'un des deux calculs a été fauté. Une façon mise en place au CESTI-LETI de déjouer cette contre-mesure est donc de faire une faute dans le DES et une faute dans le DES^{-1} afin que les deux fautes se compensent et que la comparaison réussisse malgré la contre-mesure. Avec cette méthode, le modèle d'attaquant est très réduit, car l'attaquant est capable de déterminer quel tour (*round*) du DES et du DES^{-1} il doit attaquer, ce qui limite énormément le nombre de paramètres à tester. En notant $\mathcal{P}_{\text{DES}}$ l'espace des paramètres permettant de fauter le tour visé du DES et $\mathcal{P}_{\text{DES}^{-1}}$ celui des paramètres permettant d'attaquer le tour symétrique du DES^{-1}, on a alors le taux suivant :

$$\mathcal{V} = \frac{\sum\limits_{p \in \mathcal{P}_{\text{DES}} \times \mathcal{P}_{\text{DES}^{-1}}} \sum\limits_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{\mid \mathcal{P}_{\text{DES}} \mid \cdot \mid \mathcal{P}_{\text{DES}^{-1}} \mid}$$

Généralement, les tailles des espaces $\mathcal{P}_{\text{DES}}$ et $\mathcal{P}_{\text{DES}^{-1}}$ sont similaires (car les implémentations de DES et DES^{-1} sont les mêmes), et bien plus petites que $\mid \mathcal{P}_1 \mid$, leur produit est donc encore bien plus petit que $\mid \mathcal{P}_2 \mid$.

Cet exemple montre l'intérêt des modèles d'attaquant pour déterminer $\mathcal{V}$ dans des cas complexes, notamment en cas d'injection de fautes multiples.

5.6. Conclusion : un problème ouvert

Dans ce chapitre, on a montré avec plusieurs jeux de paramètres réalistes qu'il paraît impossible d'effectuer une analyse naïve en fautes multiples dans l'état actuel de l'informatique.

Face à cette complexité, les évaluateurs tout comme les attaquants ont recours à des techniques heuristiques qui guident l'analyse vers les fautes qui paraissent *a priori* les plus à même de réussir. On a présenté une technique heuristique d'analyse des contre-mesures, et on a étendu la notion de modèle d'attaquant introduite au chapitre 4 pour quantifier le comportement des attaquants dans diverses situations.

Parallèlement, on montre que des techniques de simulation avancées telles que la compilation à la volée peuvent servir à améliorer les performances et ainsi rendre plus accessibles les attaques doubles et triples.

Pour aller plus loin et concevoir des outils à même d'injecter de nombreuses fautes indépendantes, il semble nécessaire de développer des méthodes de réduction du problème de l'injection de fautes, un problème qui reste ouvert à la fin de la thèse.

Chapitre 6

Conclusion générale

$2 + 2 = 5$ (pour des valeurs suffisamment grandes de 2).

(Houston Euler, The History of 2+2=5)

Résumé du chapitre

Ce chapitre conclut ce manuscrit par un bilan du travail effectué durant la thèse, et des perspectives pour le domaine de l'injection de fautes. On décrit l'apparition de techniques de plus en plus sophistiquées pour effectuer des attaques, et de contre-mesures proportionnées pour y répondre. Pour finir, on discute de la possibilité d'une application future de l'injection de fautes aux différents objets connectés.

Sommaire

6.1. Bilan du travail effectué

6.1.1. Approche « de bout en bout »

Tout au long de ce manuscrit, on a présenté l'approche « de bout en bout » (rappelée à la figure 6.1) pour améliorer le processus d'analyse de robustesse contre l'injection de fautes. On atteint ce but avec la séparation de ce processus entre une première phase au niveau du composant qui permet d'établir un modèle de faute, une deuxième phase

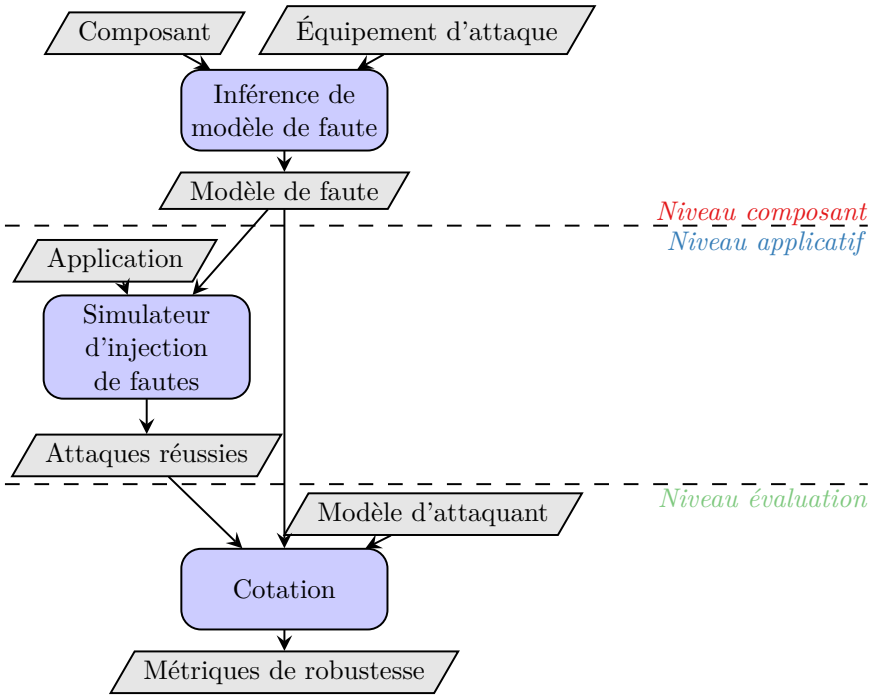


FIGURE 6.1. – Rappel de l'approche « de bout en bout »

phase au niveau applicatif qui permet d'analyser les conséquences de ce modèle de faute sur la sécurité de l'application à évaluer, et une troisième phase d'évaluation qui permet de quantifier la menace globale sur la sécurité.

La phase au niveau du composant se base sur une nouvelle notion clé : le modèle de faute probabiliste, qui décrit les fautes possibles sur un composant avec un équipement d'attaque. En support du concept de modèle de faute probabiliste, on a proposé un cadre générique adapté à notre outil pour exprimer les fautes. Pour produire les modèles de fautes probabilistes, on a introduit l'inférence de modèles de faute, une technique qui repose sur l'utilisation en séquence de plusieurs programmes spécialement conçus pour donner des informations sur les fautes effectuées, les informations obtenues avec un premier programme pouvant être réutilisées pour interpréter celles produites par le suivant.

La phase au niveau applicatif repose sur CELTIC, notre outil dynamique de simulation capable d'injecter des fautes. CELTIC effectue une campagne exhaustive d'injection de fautes en simulant toutes les exécutions fautées possibles par rapport à un modèle de faute choisi par l'utilisateur. Les modèles de faute probabilistes inférés lors de la phase au niveau du composant peuvent être réutilisés pour n'injecter que les fautes qui ont effectivement été constatées sur carte.

La phase au niveau de l'évaluation, utilise notre nouvelle métrique globale, le taux de vulnérabilité $\mathcal{V}$. Cette métrique cherche à dépasser les limitations des métriques existantes et tire parti des informations de probabilité attachées aux modèles de faute probabilistes. Le taux de vulnérabilité a fait émerger la notion clé de modèle d'attaquant pour l'injection de faute, qui cherche à quantifier l'impact des connaissances et des

capacités de l'attaquant sur la sécurité de l'application.

Enfin, on a posé un cadre préliminaire pour traiter le problème de l'injection de fautes multiples indépendantes, avec différentes pistes pour réduire sa complexité et pour permettre l'adaptation du taux de vulnérabilité à ce cas particulier via de nouveaux modèles d'attaquant.

6.1.2. Expérimentation sur carte

Tant l'inférence de modèles de faute que la validation du taux de vulnérabilité $\mathcal{V}$ ont nécessité la mise en œuvre de perturbations physiques sur plusieurs composants exécutant divers programmes. Ceci m'a permis de me familiariser avec les montages électroniques et l'équipement utilisés dans les attaques sur carte. Pour tester les programmes de détection de fautes, j'ai par ailleurs été amené à écrire un *manager* pour carte Cortex M.

6.1.3. CELTIC, un outil pour le CESTI

À l'heure actuelle la base de code de CELTIC représente plus de 22 000 lignes de code `C++11` réparties entre environ 100 classes, et plus de 6300 lignes de code `ruby`. CELTIC est un outil conçu et développé pour les évaluateurs du CESTI. L'outil est paramétrable par rapport au modèle de faute observé sur le composant, et GISL permet d'ajouter le support des divers jeux d'instructions abordés durant les évaluations. CELTIC a une dizaine d'utilisateurs au CESTI, formés via un TP de prise en main d'une demi-journée par groupes de quatre, ainsi qu'un TP d'utilisation du langage de spécification, et deux jours de formation concernant le legs du code. L'outil a été utilisé durant plusieurs évaluations réelles où il a permis de comprendre des résultats obtenus au laser.

6.1.4. Contributions

Durant la thèse, j'ai été amené à présenter l'approche « de bout en bout » dans diverses conférences et réunions : j'ai ainsi participé à une présentation inter-CESTI organisée par l'ANSSI, à une réunion du JHAS (groupe de travail « carte à puce » pour les CC), à une réunion de travail avec EMVCo (responsable de la normalisation des cartes bancaires), et même à l'occasion d'une visite du NIST (*National Institute of Standards and Technology*), et à la défendre à la conférence CARDIS 2015 [Dur+15].

J'ai également participé à l'élaboration puis au déroulement du projet SERTIF déjà évoqué au chapitre 1. Dans le cadre du projet nous avons conçu une collection de codes sécurisés [Dur+16]. Un workshop de dissémination des résultats du projet est prévu pour octobre 2016.

6.2. Perspectives

6.2.1. Poursuite des travaux

Quelles perspectives pour la poursuite des travaux ? De nouvelles expériences d'inférence de modèle de faute et de calcul de taux de vulnérabilités permettraient de

confirmer la pertinence de ces approches. De nouveaux types de programmes de détection restent encore à découvrir pour inférer efficacement des modèles de remplacement d'instruction. De plus, il serait intéressant d'établir un lien plus précis entre paramètres d'équipement et paramètres de code. Ceci permettrait de mettre en correspondance avec certitude une vulnérabilité découverte sur le code avec une attaque effectuée sur carte. On peut même imaginer de fixer les paramètres d'équipement automatiquement à partir des vulnérabilités prévues par CELTIC. Concernant CELTIC, il faut mettre en œuvre les pistes proposées au chapitre 5, pour rendre effectif le passage à l'échelle des fautes multiples. Un travail d'ingénierie permettrait également d'optimiser encore un peu les performances par l'utilisation d'une base de données. Cependant, pour traiter un grand nombre de fautes multiples, on estime qu'il faudrait un changement majeur de paradigme qui reste encore à inventer.

6.2.2. Quel futur pour l'injection de fautes ?

L'apparition des attaques par impulsion électromagnétique rend l'injection de fautes accessible à des attaquants moins puissants qui ne peuvent pas s'équiper en laser, pour des résultats presque comparables. De plus, des techniques avancées de modification de circuits à l'aide de FIB (*Focused Ion Beam*) ou de rayons X se développent, et permettent une précision exceptionnelle dans les fautes injectées (mais restent par contre inaccessibles à l'écrasante majorité des attaquants).

Les défenseurs ne restent toutefois pas inactifs face à ces menaces, et proposent des contre-mesures sophistiquées. À CARDIS 2015, Werner *et al.* ont proposé une nouvelle contre-mesure matérielle pour détecter les modifications imprévues de flot de contrôle dues aux attaques par fautes [WWM15]. Elle consiste en un circuit matériel spécialisé qui calcule une fonction de signature du chemin d'exécution à partir des instructions lues depuis la mémoire. Les auteurs annoncent un taux de détection de 99.9% en trois cycles d'horloge, pour une augmentation de 6.4% de la taille du processeur. L'augmentation du temps d'exécution semble plus variable (de 2 à 71%), mais la technique semble prometteuse à moyen terme.

Une autre démarche originale de défense contre l'injection de faute est l'idée d'un compilateur contre l'injection de faute, portée par Barry *et al.* [BCR16]. Il s'agit de rajouter des contre-mesures contre l'injection de faute au moment de la compilation du programme depuis le code source vers le format binaire. Cette approche est intéressante, car habituellement les compilateurs sont plutôt des obstacles pour les développeurs qui cherchent à se prémunir de l'injection de fautes, à cause des passes d'optimisations et de transformations du code source qui ont tendance à supprimer les contre-mesures. Il nous semble que la compilation constitue le meilleur moment du processus pour rajouter les contre-mesures. Les inconvénients majeurs de l'approche sont d'une part qu'il faut prouver que le code reste fonctionnellement correct malgré l'ajout des contre-mesures, ce qui peut s'avérer complexe, et d'autre part l'inertie des développeurs et fondeurs : les chaînes de compilation existantes doivent être modifiées pour générer les contre-mesures.

Ces nouveaux moyens de défense signifient qu'à l'image des attaques par canaux cachés, où la fuite d'information se fait de plus en plus discrète, il risque de devenir très difficile de perturber des composants sécurisés. Dans ces conditions, il est logique

d'imaginer un déplacement des attaques vers une nouvelle cible. De fait, on commence à avoir plusieurs exemples d'attaques par canaux cachés sur des ordinateurs classiques (via des ondes sonores [ST04] ou des ondes électromagnétiques [Gen+16]) et également sur les *smartphones* (attaques sur la cryptographie en boîte blanche). Les objets connectés doivent prochainement envahir notre quotidien avec l'arrivée prévue de l'*Internet of Things*. Ceux-ci réutilisent des piles de logiciels et de matériels « sur l'étagère » (COTS) et font déjà beaucoup parler d'eux pour des vulnérabilités logicielles. Une fois la sécurité logicielle assurée, il est crédible qu'ils soient ciblés par des attaques physiques. Dans ce contexte, il sera intéressant d'observer l'évolution des processus de certification. La tendance semble être à la baisse des coûts, avec l'apparition de schémas de certification simplifiés (tels que GlobalPlatform [1] pour les TEE) basés sur la vérification semi-automatisée d'un certain nombre de points prédéfinis plutôt que sur l'intervention humaine. Bien que le développement d'outil permette d'automatiser les tâches répétitives ou fastidieuses, on peut supposer que l'expertise des évaluateurs reste nécessaire pour contrer les attaques élaborées par les attaquants.

Dans le domaine de la sûreté des personnes, de nombreux systèmes, comme les voitures, les trains, et les automates industriels, voient leur sécurité assurée par l'hypothèse implicite d'une isolation physique du système. Dans notre monde actuel globalement connecté, cette hypothèse n'est plus tenable, et elle est régulièrement mise en défaut : c'est l'exemple du virus `stuxnet`, retrouvé dans des centrifugeuses iraniennes d'enrichissement d'uranium, et introduit là par des clés USB, ou encore celui de la voiture télécommandée « grandeur nature » [2], particulièrement inquiétant quand la voiture autonome se profile à l'horizon ... Les systèmes conçus pour la sûreté doivent intégrer un volet sécurité. À cette occasion, il faudrait aussi tenir compte de la possibilité d'attaque par injection de faute, un exercice d'équilibriste, car certains objectifs de sûreté entrent en conflit avec les objectifs de sécurité.

6.3. Quand 2+2=5

« *Toute technologie suffisamment avancée est indistinguable de la magie* » [3] nous disait Arthur C. Clarke. Pourtant, le domaine de l'injection de fautes est un nouveau monde, où les règles familières de la logique n'ont plus cours, et où les belles abstractions informatiques s'effondrent, renvoyées aux dysfonctionnements de leurs fondations électroniques. Si le développeur est un illusionniste, alors le laser est ce spot mal placé qui, éclairant l'artifice, en dévoile la ficelle. Cette mise en lumière provoque les huées du public, ravalant le prestidigitateur-programmeur du rang d'artiste enchanteur à celui d'escroc ridicule. Dans ce petit théâtre, l'évaluateur joue le rôle d'un technicien-machiniste, qui inspecte la scène avant la représentation, et qui s'assure que les effets spéciaux sont impénétrables. Pour les composants sécurisés, ce rôle est d'autant plus épineux que l'effet exact de l'injection de faute sur le code est difficile à décrire. Sur ces

1. http://www.globalplatform.org/

2. La vidéo de prise de contrôle par Miller et Valasek à distance d'une voiture conduite par un journaliste a fait couler beaucoup d'encre en 2015, mais Koscher *et al.* analysent cette possibilité en théorie et en pratique dès 2010 [Kos+10].

3. « *Any sufficiently advanced technology is indistinguishable from magic.* », Arthur C. Clarke, *Profiles of the Future* (1962).

coffres-forts numériques, le laser ne montre qu'un vague reflet des mécanismes, suffisant pour révéler les contours d'une supercherie, mais pas pour en exposer le cœur. Protéger le magicien d'un assistant maladroit, d'un public curieux ou d'un rival envieux n'est donc pas tâche facile pour l'évaluateur, car la défense parfaite est une chimère, et parce que bien faire son travail signifie rester dans l'ombre. Néanmoins, *the show must go on*, et aucun spectacle n'est possible sans machiniste. Il faut alors, encore et encore, donner tort à Scott Adams quand il répond à Clarke avec sarcasme : « *[. . .] [T]oute technologie suffisamment avancée ne marche pas, et personne ne sait comment la réparer* »[4].

4. « *[. . .] [A]ny sufficiently advanced technology is broken, and no one knows how to fix it.* », Scott Adams, *Stick to Drawing Comics, Monkey Brain !* (2007).

Annexe A

Attaques sur composants sécurisés

A.1. À propos des attaques logicielles...

On appelle « attaques logicielles » les attaques qui exploitent une vulnérabilité logicielle, c'est-à-dire causées par la façon dont le code du composant est écrit. L'analyse de vulnérabilité logicielle est un domaine aujourd'hui très actif en sécurité. Néanmoins, ce domaine n'a pas été l'objet de cette thèse.

On ajoutera également que, dans le cadre des cartes à puce, les attaques logicielles, bien que possibles, restent difficiles à exploiter [1]. En effet, l'attaquant n'a théoriquement **aucun** accès au code, pas même au binaire qui s'exécute sur la carte : ses entrées sont limitées au buffer APDU. La complexité globale des fonctions de sécurité est limitée (implémentation de fonctionnalités simples comme des copies de buffer ou des vérifications de code PIN, ou très spécialisées comme des primitives cryptographiques), et donc moins sujette à des erreurs de programmation. Enfin, l'évaluation garantit des bonnes pratiques de programmation (tests, gestionnaires de version, documentation).

A.2. Attaques passives

Les attaques passives (ou « non-invasives ») regroupent les attaques qui ne modifient pas le composant.

En général, on désigne par ce terme des attaques basées sur l'observation de canaux auxiliaires, ou canaux cachés, c'est-à-dire l'observation de traces émises par le composant, et qui renseignent de façon indirecte sur son comportement. Pour comprendre les attaques par canaux cachés, on peut reprendre l'analogie entre le composant sécurisé et un coffre-fort. Un coffre-fort est conçu pour s'ouvrir si l'utilisateur entre la bonne combinaison. Sa sécurité repose donc sur un secret (la combinaison). Une attaque classique sur un coffre-fort est l'utilisation d'un stéthoscope pour écouter les sons émis par le coffre-fort pendant que l'attaquant teste une combinaison. Avec une écoute attentive, il est ainsi possible de retrouver la combinaison d'un coffre-fort mécanique. On récupère le secret (la combinaison) en écoutant les canaux auxiliaires qui sont des artefacts d'implémentation dus au fonctionnement du coffre-fort.

1. On a évoqué à la section 1.2.3.2 le cas particulier des plate-formes Java Card pour lesquelles il existe des attaques.

Les attaques par canaux cachés ont donné lieu à une littérature très riche depuis la fin des années 90 dans le domaine de la carte à puce, en particulier grâce aux travaux de Kocher *et al.* [Koc96 ; KJJ98 ; KJJ99].

A.2.1. Timing Attack

La *Timing Attack* a été proposée par Kocher en 1996 [Koc96]. Elle repose sur la constatation que le temps d'exécution d'un algorithme est souvent dépendant de ses données, et que l'observation du temps d'exécution permet donc de déterminer un certain nombre d'informations sur les données, y compris quand il s'agit de données protégées en confidentialité par la carte.

Par exemple, une routine de vérification de code PIN peut être constituée d'une boucle qui vérifie chaque chiffre du code séquentiellement. Si la comparaison s'arrête dès qu'un mauvais chiffre est détecté, un attaquant obtiendra des temps de réponses différents de la carte en fonction du nombre de chiffres corrects entrés, ce qui réduit grandement l'espace des possibles en force brute.

La *Timing Attack* s'applique à de nombreux algorithmes, y compris cryptographiques. La contre-mesure pour s'en prémunir est de concevoir des algorithmes dits « équilibrés », c'est-à-dire dont le temps d'exécution ne dépend pas des données (du moins, des données à protéger en confidentialité). De telles implémentations ne doivent par exemple pas sortir d'une boucle prématurément, et ont souvent recours à des opérations fictives (*dummy operations*) sur des données inutiles pour effectuer les mêmes opérations quelles que soient les données.

A.2.2. Simple Power Analysis

La *Simple Power Analysis* consiste à mesurer directement la consommation du courant d'un composant sécurisé avec un oscilloscope durant l'exécution du code, afin d'obtenir des informations sur les instructions exécutées ou sur les données traitées (qui peuvent être des données à protéger en confidentialité).

En effet, les différentes instructions exécutées par le processeur du composant sécurisé ont chacune une consommation de courant reconnaissable. En utilisant cette signature, et si l'algorithme n'est pas équilibré, on peut obtenir des informations sur les données traitées. Messerges *et al.* appliquent par exemple l'attaque à un algorithme d'exponentiation modulaire sur carte à puce [MDS99]. L'exponentiation modulaire est utilisée dans de très nombreux algorithmes à clé publique. Le listing A.1 en donne une implémentation. Pour chaque bit d'une clé privée e, on élève au carré le message en cours de chiffrement R, puis, **si** le bit de clé est à 1, on le multiplie par le message clair d'origine M. Les figures A.1 et A.2 montrent la différence entre une élévation au carré et une multiplication. Sur une trace, comme la figure A.3, il devient alors possible de lire la clé e directement. En effet, un carré suivi d'une multiplication correspond à un bit de clé égal à 1, tandis qu'un carré isolé correspond à un bit de clé égal à 0.

Aujourd'hui, la *Simple Power Analysis* n'est typiquement plus utilisable pour obtenir des clés cryptographiques, les implémentations étant toutes protégées contre ce type d'attaque, mais elle fournit encore des informations utiles à un attaquant pour se repérer dans l'exécution du code.

FIGURE A.1. – Signature de consommation de l'instruction de mise au carré

FIGURE A.2. – Signature de consommation de l'instruction de multiplication

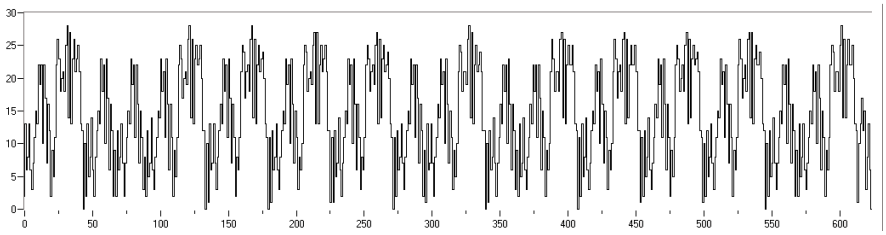


FIGURE A.3. – Trace de consommation d'un algorithme RSA. On peut lire les bits de la clé e : 1001111010111101.

</>Code source

```
for (i = k - 2; i > 0; --i) {
  // Square
  R = (R*R) % N;
  if (e[i] == 1) {
    // Mult
    R = (R*M) % N;
  }
}
```

Listing A.1. – Un extrait d'implémentation d'un algorithme d'exponentiation modulaire

A.2.3. Differential Power Analysis

Comme la *Simple Power Analysis*, la *Differential Power Analysis* (DPA) repose sur l'analyse de la consommation de courant durant l'exécution, mais à la différence de la première, la seconde repose sur un traitement statistique des acquisitions de courant pour identifier des corrélations liées aux données [KJJ99]. La méthode implique de partitionner l'ensemble des traces en différentes classes et de calculer la moyenne de chacune de ces classes. Si le partitionnement est décorrélé des mesures contenues dans les traces, les moyennes font converger vers zéro avec l'augmentation du nombre de traces dans chaque classe. À l'inverse, si le partitionnement est corrélé à une donnée (par exemple, un secret cryptographique), les moyennes de chaque classe vont tendre vers une valeur non nulle. Les attaques DPA s'effectuent donc à des moments d'un algorithme cryptographique où peu de bits de secret sont manipulés. Les mesures sont partitionnées suivant une hypothèse de clé, et les hypothèses de clé sont ainsi testées exhaustivement jusqu'à trouver celle qui fait converger les moyennes de chaque classe vers une valeur non nulle : cette hypothèse correspond à la bonne clé. En 2004, Brier *et al.* proposent la *Correlation Power Analysis* [BCO04]. Cette méthode vise à améliorer la DPA en formalisant le facteur de corrélation à l'aide de la corrélation de Pearson, et en explicitant un modèle de fuite basé sur la distance de Hamming.

A.2.4. Attaques template

Les attaques *template* sont présentées en 2002 par Chari *et al.* comme « la forme la plus forte d'attaque par canaux cachés possible, au sens de la théorie de l'information » [2] [CRR03]. Cette classe d'attaques repose sur la constatation que la *Differential Power Analysis* utilise un grand nombre d'échantillons de consommation de la carte attaquée dans le but de séparer le bruit du signal utile via des moyennes sur tous les signaux. Les attaques *template* prennent le contrepied de cette approche en cherchant à modéliser très précisément le bruit afin de pouvoir tirer le maximum d'information d'un

2. « *We present* template attack*, the strongest form of side channel attack possible in an information theoretic sense.* » [CRR03]

unique échantillon de consommation de la carte attaquée. Ce type d'attaque s'effectue donc en deux étapes : dans une première étape, dite de profilage, l'attaquant établit le modèle de fuite du composant. Pour ce faire, il doit disposer d'un composant identique à celui qu'il attaque, avec la possibilité de choisir les paramètres cryptographiques, par exemple une clé cryptographique. Pour établir le modèle, il effectue un premier lot d'expériences en faisant varier les paramètres qu'on peut choisir. Il faut ensuite détecter des points d'intérêts, qui sont des points particuliers sur les courbes de consommation qui correspondent à des instants où l'algorithme dépend de quelques bits des données cryptographiques. On peut ensuite faire varier exhaustivement ces bits pour établir précisément le profil de fuite. On reproduit cette première étape avec d'autres points d'intérêts jusqu'à avoir établi le profil de toute la clé. Dans la seconde étape, sur une trace du composant attaqué (que l'attaquant ne contrôle pas), il suffit d'identifier à quel profil chaque partie de la courbe correspond.

Ce type d'attaque est extrêmement puissant, mais suppose un grand pouvoir à l'attaquant pour pouvoir réaliser la première étape. Des variantes des attaques *template* proposent d'essayer d'établir le modèle de bruit sans connaître les paramètres cryptographiques pendant la phase de profilage [Kar+09].

A.3. Attaques invasives

Les attaques dites invasives ont pour caractéristique la modification physique permanente du matériel attaqué.

On les classe en deux catégories suivant qu'elles détruisent le composant sécurisé sur lequel on les applique ou non. Pour le premier cas, on parle d'attaques destructrices.

A.3.1. Décapsulation de composants

La décapsulation de composants consiste à enlever l'emballage du composant sécurisé afin d'avoir un accès direct au silicium.

Elle s'effectue à l'aide d'un acide nitrique fumant très concentré, à 60 °C, sur le composant déjà chauffé [Bec98 ; KK99]. Malgré la dangerosité de l'opération, elle ne nécessite pas de connaissances chimiques ou électroniques supérieures à celles qu'on peut acquérir en suivant les travaux pratiques du secondaire [Sko05].

Cette opération est généralement réalisée de façon non destructrice pour le composant, c'est-à-dire que le composant est toujours opérationnel après la décapsulation. Cette attaque est très utile en ce qu'elle permet ensuite d'autres attaques : dans les attaques invasives, la rétroconception matérielle nécessite la décapsulation, de même que le micro-sondage (le bon fonctionnement du composant est alors important). Dans les attaques semi-invasives, l'attaque par laser est rendue possible par la décapsulation, tandis que les attaques électromagnétiques sont grandement facilitées.

A.3.2. Rétroconception matérielle

La rétroconception matérielle est l'ensemble des opérations de démontage d'un composant sécurisé visant à comprendre sa conception. C'est une attaque destructrice, car elle nécessite la suppression successive de toutes les couches (métalliques ou de silice)

que comprend le composant, qui sont photographiées avant leur élimination. À un niveau de détail suffisant, elle permet l'observation directe des transistors qui constituent la logique du composant et de ses différentes mémoires. Le code du programme du composant sécurisé peut alors être lu directement dans la ROM, ce qui permet sa rétroconception logicielle [Sko05]. La rétroconception matérielle est malheureusement très difficile sur le matériel actuel, notamment à cause de la finesse de gravure, et parce que le contenu de la ROM est généralement chiffré dans un composant sécurisé.

A.3.3. Micro-sondage

Les attaques par micro-sondage sont des attaques invasives non destructrices qui visent à poser des sondes sur des fils très spécifiques du composant pour pouvoir observer le courant qui passe sur ces fils.

Associées à un peu de rétroconception (d'un autre exemplaire du composant, la rétroconception étant destructrice), les attaques par microsondage permettent par exemple de se connecter aux sorties de boîtes-S d'une implémentation matérielle de DES, ce qui peut permettre de lire directement les sous-clés.

Pour des composants à la finesse de gravure inférieure à 0.35 µm, on ne peut pas utiliser directement des électrodes, il faut avoir recours à un *Focused Ion Beam* (FIB) qui dispose d'une résolution de 10 nm [Sko11].

Annexe B

Inférence de modèle de faute sur la carte A

B.1. Dispositif Expérimental

La carte A est une carte sécurisée [1] qui possède un cœur Cortex-M3 (ARMv7-M). Elle est cadencée par une horloge externe à 4.186 MHz. Elle propose plusieurs contre-mesures matérielles (détecteur de *glitches*, détecteurs de laser). Pour la carte A, on a réutilisé l'injecteur électromagnétique utilisé pour la carte C. Le programme de détection de faute est le même que le premier utilisé pour la carte C : le programme de détection sur la mémoire non volatile. On a cependant utilisé un tampon de demi-mots (2 octets) au lieu d'un seul octet, la justification étant que les lectures d'instructions se font par demi-mot. Dans la pratique, ce choix a le gros inconvénient qu'il y a 2^{16} valeurs à tester, ce qui n'est pas réalisable en pratique. On a donc choisi aléatoirement un échantillon de 300 demi-mots (dans l'intervalle $[0, 65\,535]$), dont les valeurs spéciales 0 et `0xFFFF`, sur lequel faire les tests.

B.2. Phase d'initialisation

La phase d'initialisation nous a permis de déterminer les positions (x, y) à la surface de la carte, telles que des attaques EEPROM ont lieu suffisamment souvent (sans perturbation RAM ou registre par ailleurs). La figure B.1 indique qualitativement l'étendue de ces zones (en rouge). Durant la phase d'initialisation, on fait varier l'angle θ entre l'injecteur et la carte : on observe que pour les valeurs $(-90°, 90°)$, on obtient essentiellement des *bitreset*, tandis que pour les valeurs $(0°, 180°)$, on obtient essentiellement des *bitsets*. Ces résultats sont cohérents avec ceux de la littérature [Ord+15]. Suite à cette phase d'initialisation, on choisit :

$$\mathcal{P}_0 = \{(\theta = -90°, x = x_0, y = y_0, z = z_0, d = t_0 + 10k) \mid k \in \{0, \ldots, 39\}\}$$

où (x_0, y_0) sont des valeurs fixées dans la zone rouge de la figure B.1, z_0 est une constante permettant d'obtenir un niveau d'énergie suffisant pour avoir des fautes couramment, et t_0 permet d'attaquer le début du tampon EEPROM.

1. C'est-à-dire qu'elle a reçu un certificat EAL4+ dans le cadre d'une évaluation CC.

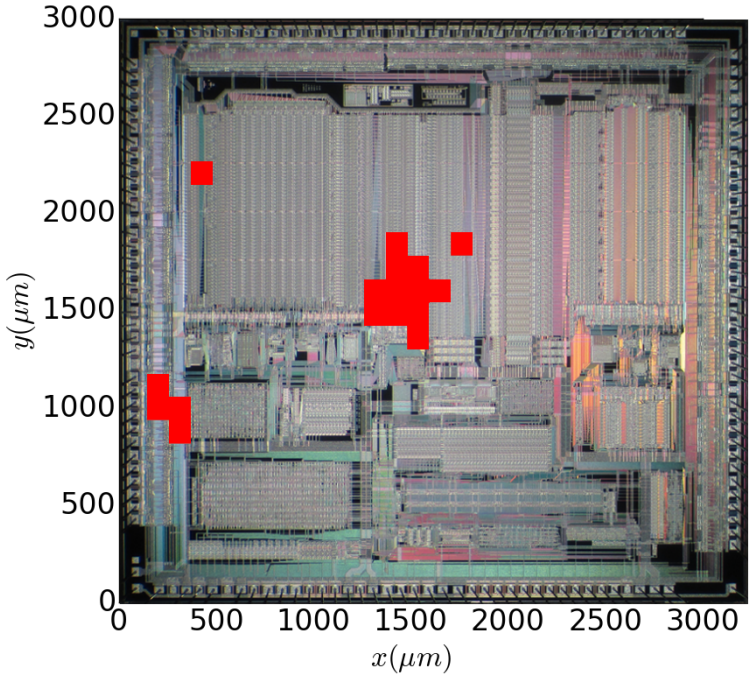


FIGURE B.1. – Cartographie des fautes pour la carte A

B.3. Phase itérative

Pour chaque couple (p, a) de paramètre d'équipement et de demi-mot d'entrée du programme de détection, on a effectué 30 répétitions, pour un total de $30 \times 40 \times 300 = 360\,000$ essais sur 10 jours. Le type de faute observé était majoritairement la modification d'une case du tampon depuis la valeur a vers une valeur b. Exceptionnellement, on a pu avoir deux demi-mots consécutifs mis à 0. Le traitement statistique des résultats bruts nous a permis d'extraire la figure B.2, une carte de chaleur qui donne la probabilité qu'un demi-mot a dans le tampon soit remplacé par le demi-mot b. Plus la probabilité est forte en un point (a, b), plus le point est foncé sur la carte de chaleur. On observe deux phénomènes, illustrés par les lignes en pointillés sur la figure B.2 :

— La ligne horizontale d'équation $y = 0$ (en bleu) indique que la probabilité est forte d'obtenir $b = 0$, quel que soit a non nul [2].
— La ligne oblique d'équation $x = y$ (en rouge) indique que la probabilité d'obtenir une valeur b proche de a est forte [3].

2. On observe des « trous » dans la ligne dus au fait qu'on n'a testé que 300 valeurs sur les 65 535 possibles.
3. Obtenir la même valeur n'est pas considéré comme une faute, et donc pas reporté sur le schéma, il s'agit donc de valeurs proches.

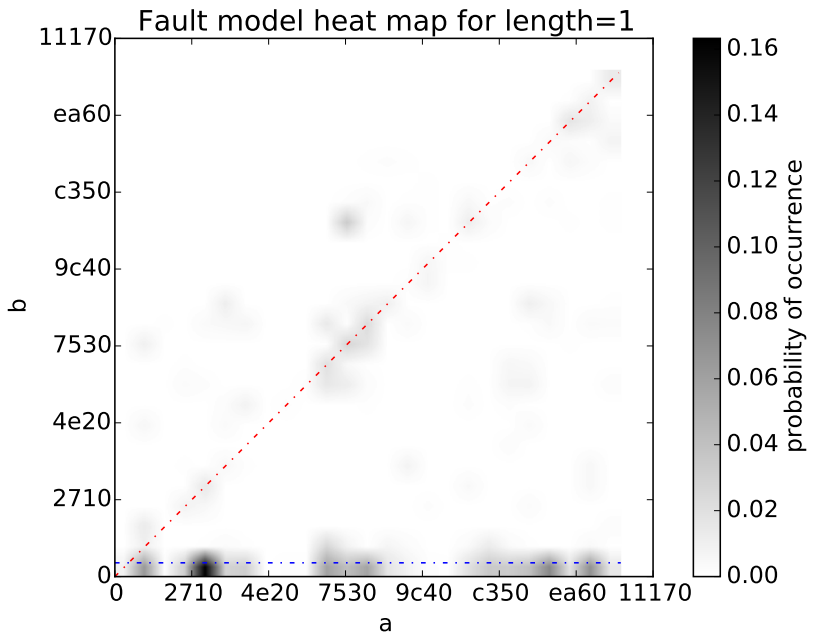


FIGURE B.2. – Carte de chaleur pour la carte A

B.4. Généralisation

On généralise le modèle brut obtenu de 2 façons :

— On généralise les comportements obtenus sur les 300 valeurs aux 65 535 demi-mots.

— On généralise l'espace de paramètres d'équipement à :

$$\mathcal{P} = \{(-90^\circ, x_0, y_0, z_0, t_0 + j\delta) \mid j \in \mathbb{N}, \delta = 720\,\text{ns}\}$$

à l'aide d'une technique d'exploitation de la périodicité suivant la dimension temporelle. Pour ce faire, on observe la probabilité d'injection de faute en fonction du temps (voir B.3) et on remarque une périodicité $\delta = 720\,\text{ns}$, associée à une relation entre le délai d d'attaque et l'indice de début d'attaque dans le tampon EEPROM. Précisément, si d est compris entre $t_0 + j\delta$ et $t_0 + (j+1)\delta$, alors le $j^{\text{ème}}$ demi-mot du tampon est attaqué. On se permet cette généralisation car les lectures mémoire se répètent séquentiellement, il est donc naturel qu'elles produisent un comportement périodique.

On obtient au final les ensembles suivants :

$$\{(d = t_0 + k\delta) \underset{f_0}{\rightsquigarrow} (i = k, \text{read}_m(\text{EEPROM}[x_k], a, 0)),\ k \in \mathbb{N} \wedge a \neq 0\}$$

$$\{(d = t_0 + k\delta) \underset{f_{1\%}}{\rightsquigarrow} (i = k, \text{read}_m(\text{EEPROM}[x_k], a, b)),\ k \in \mathbb{N} \wedge a \neq 0 \wedge d(a,b) \leqslant 1\%\}$$

$$\{(d = t_0 + k\delta) \underset{f_{20\%}}{\rightsquigarrow} (i = k, \text{read}_m(\text{EEPROM}[x_k], a, b)),\ k \in \mathbb{N} \wedge a \neq 0 \wedge d(a,b) \leqslant 20\%\}$$

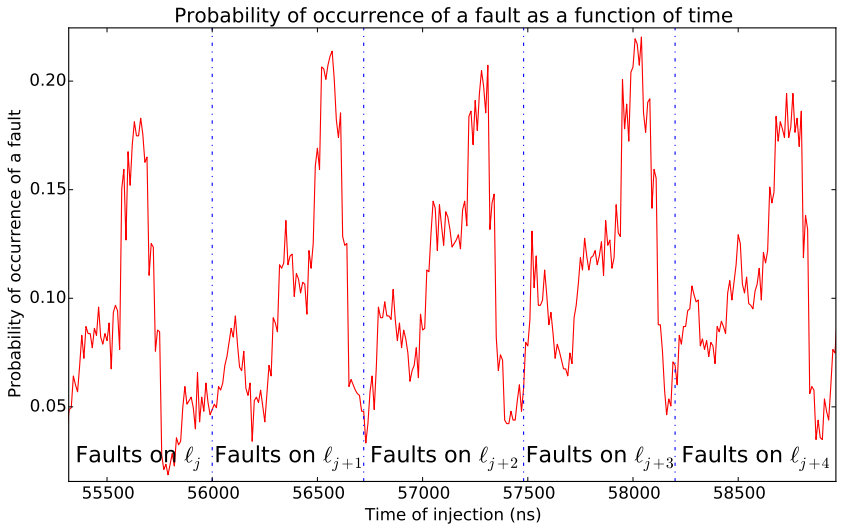


FIGURE B.3. – Péridiocité de la probabilité d'injecter une faute en fonction du temps

$$\{(d = t_0 + k\delta) \underset{f_{\text{rand}}}{\rightsquigarrow} (i = k, \text{read}_m(\text{EEPROM}[x_k], a, b)),\ k \in \mathbb{N} \wedge a \neq 0 \wedge d(a,b) > 20\%\}$$

$$\{(d = t_0 + k\delta) \underset{f_{\text{double}}}{\rightsquigarrow} (i = (k, k+1), (\text{read}_m(\text{EEPROM}[x_k], a_0, 0), \text{read}_m(\text{EEPROM}[x_k + 1], a_1, 0))),\ k \in \mathbb{N} \wedge (a_0, a_1) \neq (0,0)\}$$

On rappelle le modèle de faute obtenu dans le tableau B.1 (identique au tableau 3.5 page 88).

Faute	Probabilité
f_0	4.8%
$f_{1\%}$	1.8%
$f_{20\%}$	1.6%
f_{rand}	1.3%
f_{double}	0.5%

TABLE B.1. – Modèle de faute probabiliste pour la carte A

Annexe C

Explication de quelques attaques

Dans cette annexe, on explique plusieurs attaques sélectionnées parmi les résultats de la campagne d'injection de fautes avec un modèle de remplacement exhaustif d'un octet du code. La campagne a été présentée dans l'exemple 4.7 page 118 du chapitre 4. Les attaques sélectionnées apparaissent en gras dans la table 4.4 page 118.

C.1. Rappel de l'exemple

On redonne le code source de l'implémentation évaluée dans le listing C.1. On fournit également les listings C.2 et C.3 qui correspondent à l'assembleur ARMv7M des fonctions `byteArrayCompare` et `verifyPIN`.

Dans la suite, on explique les attaques aux adresses données dans la table C.1.

Adresse	Nombre d'attaques réussies
0x41cb	8
0x41f3	8
0x41e9	8
0x41be	7

TABLE C.1. – Attaques expliquées dans cette annexe

</>Code source

```
extern SBYTE g_ptc;
extern BOOL g_authenticated;
extern UBYTE g_userPin[PIN_SIZE];
extern UBYTE g_cardPin[PIN_SIZE];

BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size)
{
  int i;
  BOOL status = BOOL_FALSE;
  BOOL diff = BOOL_FALSE;
  for (i = 0; i < size; i++) {
    if (a1[i] != a2[i]) {
      diff = BOOL_TRUE;
    }
  }
  if (i != size) {
    countermeasure();
  }
  if (diff == BOOL_FALSE) {
    status = BOOL_TRUE;
  } else {
    status = BOOL_FALSE;
  }
  return status;
}

BOOL verifyPIN_2()
{
  g_authenticated = BOOL_FALSE;

  if (g_ptc > 0) {
    if (byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE) == BOOL_TRUE) {
      g_ptc = 3;
      g_authenticated = BOOL_TRUE; // Authentication();
      return BOOL_TRUE;
    } else {
      g_ptc--;
      return BOOL_FALSE;
    }
  }
  return BOOL_FALSE;
}
```

Listing C.1. – Code source de l'implémentation testée

</>Code assembleur

```
041a8 <byteArrayCompare>:
 41a8:        b570             push        {r4, r5, r6, lr}
 41aa:        2455             movs        r4, #85        ; 0x55
 41ac:        2300             movs        r3, #0
 41ae:        e005             b.n        41bc <byteArrayCompare+0x14>
 41b0:        5cc5             ldrb        r5, [r0, r3]
 41b2:        5cce             ldrb        r6, [r1, r3]
 41b4:        42b5             cmp        r5, r6
 41b6:        d000             beq.n        41ba <byteArrayCompare+0x12>
 41b8:        24aa             movs        r4, #170        ; 0xaa
 41ba:        1c5b             adds        r3, r3, #1
 41bc:        4293             cmp        r3, r2
 41be:        dbf7             blt.n        41b0 <byteArrayCompare+0x8>
 41c0:        4293             cmp        r3, r2
 41c2:        d001             beq.n        41c8 <byteArrayCompare+0x20>
 41c4:        f000 f828        bl        4218 <countermeasure>
 41c8:        2c55             cmp        r4, #85        ; 0x55
 41ca:        d001             beq.n        41d0 <byteArrayCompare+0x28>
 41cc:        2055             movs        r0, #85        ; 0x55
 41ce:        bd70             pop        {r4, r5, r6, pc}
 41d0:        20aa             movs        r0, #170        ; 0xaa
 41d2:        bd70             pop        {r4, r5, r6, pc}
```

Listing C.2. – Code assembleur de la fonction `byteArrayCompare`

</>Code assembleur

```
041d4 <verifyPIN_2>:
 41d4:        b570             push        {r4, r5, r6, lr}
 41d6:        4d0c             ldr        r5, [pc, #48]        ; (4208 <verifyPIN_2+0x34>)
 41d8:        4c0c             ldr        r4, [pc, #48]        ; (420c <verifyPIN_2+0x38>)
 41da:        2055             movs        r0, #85        ; 0x55
 41dc:        7028             strb        r0, [r5, #0]
 41de:        f994 0000        ldrsb.w        r0, [r4]
 41e2:        2800             cmp        r0, #0
 41e4:        dd09             ble.n        41fa <verifyPIN_2+0x26>
 41e6:        2204             movs        r2, #4
 41e8:        4909             ldr        r1, [pc, #36]        ; (4210 <verifyPIN_2+0x3c>)
 41ea:        480a             ldr        r0, [pc, #40]        ; (4214 <verifyPIN_2+0x40>)
 41ec:        f7ff ffdc        bl        41a8 <byteArrayCompare>
 41f0:        28aa             cmp        r0, #170        ; 0xaa
 41f2:        d004             beq.n        41fe <verifyPIN_2+0x2a>
 41f4:        7820             ldrb        r0, [r4, #0]
 41f6:        1e40             subs        r0, r0, #1
 41f8:        7020             strb        r0, [r4, #0]
 41fa:        2055             movs        r0, #85        ; 0x55
 41fc:        bd70             pop        {r4, r5, r6, pc}
 41fe:        2003             movs        r0, #3
 4200:        7020             strb        r0, [r4, #0]
 4202:        20aa             movs        r0, #170        ; 0xaa
 4204:        7028             strb        r0, [r5, #0]
 4206:        bd70             pop        {r4, r5, r6, pc}
 4208:        10008014         .word        0x10008014
 420c:        10008015         .word        0x10008015
 4210:        1000801b         .word        0x1000801b
 4214:        10008017         .word        0x10008017
```

Listing C.3. – Code assembleur de la fonction `verifyPIN`

C.2. Attaques assimilables à des inversions de test

Les attaques aux adresses `0x41cb` et `0x41f3` correspondent à des fautes qui modifient une instruction de saut conditionnel en un saut conditionnel de condition « inverse ». Ceci permet d'effectuer le saut au lieu de continuer en séquence. Ce type d'attaque n'est pas réalisable avec le modèle de faute NOP, contrairement au cas où il s'agit de ne pas effectuer un saut normalement emprunté pendant l'exécution nominale Au niveau source, ce type d'attaque est équivalent à une inversion de test. Par exemple, la suppression de l'instruction de saut à l'adresse `0x41cb` correspond à l'inversion du test de la ligne 19, qui permet de mettre `BOOL_TRUE` dans `status`. De même, la suppression de l'instruction de saut à l'adresse `0x41f3` correspond à l'inversion du test de la ligne 32, qui permet d'effectuer le code d'authentification aux lignes 33-35. En ARMv7-M, il existe 14 instructions de branchement conditionnel, et quel que soit l'état du processeur, 7 d'entre-elles sont toujours vérifiées. Il existe de plus 1 instruction de branchement inconditionnel, ce qui explique que 8 valeurs permettent de réussir l'attaque.

C.3. Attaques de remplacement d'une instruction par un saut

L'attaque à l'adresse `0x41e9` a un principe proche des attaques aux adresses `0x41cb` et `0x41f3`, sauf que l'instruction visée n'est pas initialement un saut conditionnel. L'écrasement de l'instruction à l'adresse `0x41e8`, qui a la particularité que son opérande `0x09` est interprété comme un décalage de `22` [1], qui détermine une adresse de branchement à `0x41fe` si l'on remplace son code opération par l'un des 8 codes d'instruction de saut effectuables dans l'état du processeur à l'adresse `0x41e8`. Or, les instructions à partir de l'adresse `0x41fe` correspondent au code d'authentification des lignes 33 à 35 du source. Au niveau source, c'est l'équivalent d'ajouter un `goto` juste avant l'appel à `byteArrayCompare`, vers la ligne 33.

En supposant un modèle de faute compatible avec l'écriture des 8 valeurs qui fonctionnent, on obtient une classe de vulnérabilité potentiellement très dangereuse et très difficile à détecter lors d'une revue manuelle (puisqu'elle dépend de l'encodage binaire).

C.4. Attaques de modification de l'adresse d'un saut

L'attaque à l'adresse `0x41be` fait intervenir l'instruction `blt.n 41b0` qui permet normalement de passer au prochain tour de boucle si `i < size`, ou de quitter la boucle dans le cas contraire. L'attaque fonctionne en modifiant l'opérande de cette instruction de saut conditionnel pour la faire pointer à une adresse qui conduit à la victoire. Plusieurs valeurs sont possibles (d'où les 7 fautes possibles), mais la plus impressionnante est la valeur `0x20`, qui se traduit par un saut de l'adresse `0x41be` à l'adresse `0x4202`, c'est-à-dire un passage de la ligne 11 à la ligne 33 du code source, en passant directement de `byteArrayCompare` à `verifyPIN`! De là, on exécute les trois instructions suivantes (fin de `verifyPIN`) :

1. Par le jeu des calculs de décalages ARM.

```
4202:  movs r0, #170
4204:  strb r0, [r5, #0]
4206:  pop {r4, r5, r6, pc}
```

Puis, l'instruction de retour de `verifyPIN` ramène à la ligne 32 du source, juste après l'appel à `byteArrayCompare` (on a changé de fonction sans changer de contexte, en particulier on n'avait pas modifié l'adresse de retour de l'appel à `byteArrayCompare`), ce qui correspond à l'adresse `0x41f0` de l'assembleur.

De là, comme l'instruction à l'adresse `0x4202` a mis la constante `BOOL_TRUE` dans le registre `r0`, on entre dans la branche de victoire de la condition ligne 32 et on exécute le code d'authentification des lignes 33 à 35. Cette attaque interprocédurale est extrêmement difficile à détecter lors de la revue manuelle (cela supposerait de vérifier manuellement les modifications possibles des opérandes des sauts).

Acronymes

ADL *Architecture Description Language* ou « langage de spécification d'architecture ». 98

ANSSI Agence Nationale pour la Sécurité des Systèmes d'Information. 9, 10, 53

BSI Bundesamt für Sicherheit in der Informationstechnik. 10

CC Critères Communs. 9, 10, 11, 28, 87, 110, 116, 120, ix

CEM *Common methodology for information technology security evaluation. 11, 15, 35*

CESTI *Centre d'Évaluation de la Sécurité des Technologies de l'Information. 10, 22, 25, 29, 42, 54, 59, 69, 95, 96, 97, 103, 116, 119, 131*

CISC Complex Instruction Set Computer. 131, 132, 134, 135

COTS *Components Off the Shelf* ou « composants sur l'étagère ». 9, 20, 41

CSPN Certification Sécuritaire de Premier Niveau. 9

EAL *Evaluation Assurance Level* ou « niveau d'assurance d'évaluation ». 11, 87, ix

FIB *Focused Ion Beam. vi*

FISSC Fault Injection and Simulation Secure Collection. 98, 109, 111, 158

JIL *Joint Interpretation Library. xiii, 15, 16, 35, 126*

PP Protection Profile *ou « profil de protection ». 11*

RISC Reduced Instruction Set Computer. 131, 134, 135

SAR *Security Assurance Requirement* ou « exigence d'assurance de sécurité ». 11, 12, 15, 25

SFR *Security Functional Requirement* ou « exigence fonctionnelle de sécurité ». 11

SIM *Subscriber Identity Module. 5, 7, 8*

ST Security Target *ou « cible de sécurité ». 10*

TEE Trusted Execution Environment. 5, 8, 11

TNT Télévision Numérique Terrestre. 8

TOE *Target Of Evaluation* ou « cible de l'évaluation ». 10, 15

TPM *Trusted Platform Module. 5*

Bibliographie

[ABI15] ABI Research. *8.8 Billion Smart Cards Shipped in 2014 Driven by Growth in the Banking and SIM Card Markets*. https://www.abiresearch.com/press/88-billion-smart-cards-shipped-in-2014-driven-by-g/. En ligne ; accédé le 25 juillet 2016. Jan. 2015 (cf. p. 7).

[AK98] Ross Anderson et Markus G. Kuhn. "Low cost attacks on tamper resistant devices". English. In : *Security Protocols*. Sous la dir. de Bruce Christianson, Bruno Crispo, Mark Lomas et Michael Roe. T. 1361. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, p. 125–136. isbn : 978-3-540-64040-0. doi : 10.1007/BFb0028165. url : http://dx.doi.org/10.1007/BFb0028165 (cf. p. 19).

[And09] Philippe Andouard. "Outils d'aide à la recherche de vulnérabilités dans l'implantation d'applications embarquées sur carte à puce". Thèse de doct. Bordeaux 1, 2009 (cf. p. 40, 42, 44–52, 106, 111).

[Avi76] Algirdas A. Avižienis. "Fault-tolerant systems". In : *IEEE Transactions on Computers* 12 (1976), p. 1304–1312 (cf. p. 37, 47).

[Bar+05] Raul Barbosa, Jonny Vinter, Peter Folkesson et Johan Karlsson. "Assembly-level pre-injection analysis for improving fault injection efficiency". In : *European Dependable Computing Conference*. Springer. 2005, p. 246–262 (cf. p. 152).

[Bar+11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud et Santiago Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer". English. In : *Advances in Cryptology – CRYPTO 2011*. Sous la dir. de Phillip Rogaway. T. 6841. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, p. 71–90. isbn : 978-3-642-22791-2. doi : 10.1007/978-3-642-22792-9_5. url : http://dx.doi.org/10.1007/978-3-642-22792-9_5 (cf. p. 41).

[Bar+14] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire et Jean-Christophe Zapalowicz. "Synthesis of Fault Attacks on Cryptographic Implementations". In : *CCS '14 : Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Scottsdale, Arizona, USA : ACM, 2014, p. 1016–1027. isbn : 978-1-4503-2957-6. url : http://doi.acm.org/10.1145/2660267.2660304 (cf. p. 23, 41–43, 45–49, 51–53).

[BCO04] Eric Brier, Christophe Clavier et Francis Olivier. "Correlation power analysis with a leakage model". In : *Cryptographic Hardware and Embedded Systems-CHES 2004*. Springer, 2004, p. 16–29 (cf. p. iv).

[BCR16] Thierno Barry, Damien Couroussé et Bruno Robisson. “Compilation of a Countermeasure Against Instruction-Skip Fault Attacks”. In : *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*. ACM. 2016, p. 1–6 (cf. p. 166).

[BDL97] Dan Boneh, Richard A. DeMillo et Richard J. Lipton. “On the importance of checking cryptographic protocols for faults”. In : *Advances in Cryptology—EUROCRYPT'97*. Springer. 1997, p. 37–51 (cf. p. 18).

[BE14] Tomislav Burić et Neven Elezović. “Asymptotic expansions of the binomial coefficients”. In : *Journal of applied mathematics and computing* 46.1-2 (2014), p. 135–145 (cf. p. 143).

[Bec98] Friedrich Beck. *Integrated Circuit Failure Analysis : A guide to preparation techniques*. John Wiley & Sons, 1998 (cf. p. v).

[Bel05] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In : *USENIX Annual Technical Conference, FREENIX Track*. 2005, p. 41–46 (cf. p. 42).

[Ben+98] Alfredo Benso, Maurizio Rebaudengo, Leonardo Impagliazzo et Pietro Marmo. “Fault-list collapsing for fault-injection experiments”. In : *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*. Jan. 1998, p. 383–388. doi : 10.1109/RAMS.1998.653808 (cf. p. 152).

[Ber+12] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky et Jean-François Lalande. “High level model of control flow attacks for smart card functional security”. In : *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*. IEEE. 2012, p. 224–229. isbn : 978-1-4673-2244-7 (cf. p. 40, 42–44, 46–52, 128).

[Ber+14] Maël Berthier, Julien Bringer, Hervé Chabanne, Thanh-Ha Le, Lionel Rivière et Victor Servant. “Idea : Embedded Fault Injection Simulator on Smartcard”. In : *ESSoS*. 2014, p. 222–229 (cf. p. 41–43, 45–53, 106, 117).

[BIL11] Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet. “Combined software and hardware attacks on the java card control flow”. In : *Smart Card Research and Advanced Applications*. Springer, 2011, p. 283–296 (cf. p. 22).

[BJ15] Jakub Breier et Dirmanto Jap. “Testing Feasibility of Back-Side Laser Fault Injection on a Microcontroller”. In : *Proceedings of the WESS'15 : Workshop on Embedded Systems Security*. ACM. 2015, p. 5 (cf. p. 20).

[BOS03] Johannes Blömer, Martin Otto et Jean-Pierre Seifert. “A new CRT-RSA algorithm secure against bellcore attacks”. In : *CCS '03*. Washington D.C., USA : ACM, 2003, p. 311–320. isbn : 1-58113-738-9. doi : 10.1145/948109.948151. url : http://doi.acm.org/10.1145/948109.948151 (cf. p. 23).

[BR10] Gogul Balakrishnan et Thomas Reps. “WYSINWYX : What You See Is Not What You eXecute”. In : *ACM Transactions on Programming Languages and Systems* 32 (6 août 2010), 23 :1–23 :84. issn : 0164-0925 (cf. p. 44).

[Bra+04] Gunnar Braun, Achim Nohl, Andreas Hoffmann, Oliver Schliebusch, Rainer Leupers et Heinrich Meyr. “A universal technique for fast and flexible instruction-set architecture simulation”. In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.12 (2004), p. 1625–1639 (cf. p. 155).

[BTG10] Guillaume Barbu, Hugues Thiebeauld et Vincent Guerin. “Attacks on Java Card 3.0 Combining Fault and Logical Attacks”. Anglais. In : *Smart Card Research and Advanced Application. 9th IFIP WG 8.8/11.2 International Conference*. Sous la dir. de Dieter Gollmann, Jean-Louis Lanet et Julien Iguchi-Cartigny. T. 6035. Lecture Notes in Computer Science / Security & Cryptology. Passau, Allemagne : Springer, avr. 2010, p. 148–163. doi : 10.1007/978-3-642-12510-2_11. url : http://hal.archives-ouvertes.fr/hal-00692165 (cf. p. 22).

[Cam+08] Drew Campbell, Jason Harper, Vinodhkumar Natham, Funian Xiao et Raji Sundararajan. “A compact high voltage nanosecond pulse generator”. In : *Proc. ESA AME* (2008), p. 1–12 (cf. p. 20).

[Cat86] Jim Cathey. “COM : An 8080 Simulator for the MC68000”. In : *Dr. Dobb’s J.* 11.1 (jan. 1986), p. 76–82. issn : 1044-789X. url : http://dl.acm.org/citation.cfm?id=11948.11952 (cf. p. 103).

[CCR12a] The CCRA Management Committee. *Common Criteria for Information Technology Security Evaluation, part 2 : Security functional components*. Sept. 2012. url : http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R4.pdf (visité le 03/05/2013) (cf. p. 11).

[CCR12b] The CCRA Management Committee. *Common Criteria for Information Technology Security Evaluation, part 3 : Security Assurance Components*. Sept. 2012. url : http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R4.pdf (visité le 03/05/2013) (cf. p. 11, 12).

[CDE08] Cristian Cadar, Daniel Dunbar et Dawson R. Engler. “KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In : *OSDI*. 2008, p. 209–224 (cf. p. 41, 46).

[Chr+13] Maria Christofi, Boutheina Chetali, Louis Goubin et David Vigilant. “Formal verification of a CRT-RSA implementation against fault attacks”. In : *Journal of Cryptographic Engineering* 3.3 (2013), p. 157–167 (cf. p. 23).

[Chr13] Maria Christofi. “Preuves de sécurité outillées d’implémentation cryptographiques”. Thèse de doct. Laboratoire PRiSM, Université de Versailles Saint Quentin-en-Yvelines, France, 2013 (cf. p. 23, 40, 42–49, 51, 52).

[CMS98] João Carreira, Henrique Madeira et João Gabriel Silva. “Xception : A technique for the experimental evaluation of dependability in modern computers”. In : *Software Engineering, IEEE Transactions on* 24.2 (1998), p. 125–136 (cf. p. 37).

[Cow+03] Crispin Cowan, Steve Beattie, John Johansen et Perry Wagle. “Pointguard TM : protecting pointers from buffer overflow vulnerabilities”. In : *Proceedings of the 12th conference on USENIX Security Symposium*. T. 12. 2003, p. 91–104 (cf. p. 21).

[Cow+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang et Heather Hinton. “StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” In : *Usenix Security*. T. 98. 1998, p. 63–78 (cf. p. 21).

[Cra+97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson et Mario Wolczko. “Compiling java just in time”. In : *IEEE Micro* 17.3 (1997), p. 36–43 (cf. p. 155).

[CRR03] Suresh Chari, Josyula R. Rao et Pankaj Rohatgi. “Template attacks”. In : *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2003, p. 13–28 (cf. p. iv).

[CUL89] Craig Chambers, David Ungar et Elgin Lee. “An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes”. In : *ACM Sigplan Notices*. T. 24. 10. ACM. 1989, p. 49–70 (cf. p. 155).

[DH76] Whitfield Diffie et Martin E. Hellman. “New directions in cryptography”. In : *Information Theory, IEEE Transactions on* 22.6 (1976), p. 644–654 (cf. p. 36).

[Dur+15] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas et Jessy Clédière. “From Code Review to Fault Injection Attacks : Filling the Gap”. In : *14th Smart Card Research and Advanced Application Conference (CARDIS’15)*. Springer, 2015 (cf. p. 67, 70, 87, 165).

[Dur+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen et Philippe de Choudens. “FISSC : a Fault Injection and Simulation Secure Collection”. In : *Computer Safety, Reliability and Security – 35th International Conference SAFECOMP 2016*. Lectures Notes in Computer Science. Springer, 2016 (cf. p. 98, 152, 165).

[Fau08] Olivier Faurax. “Évaluation par simulation de la sécurité des circuits face aux attaques par faute”. Thèse de doct. Université de la Méditerranée-Aix-Marseille II, 2008 (cf. p. 39, 42, 43, 45, 46, 48–52, 111).

[FLV12] Pierre-Alain Fouque, Delphine Leresteux et Frédéric Valette. “Using faults for buffer overflow effects”. In : *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, p. 1638–1639 (cf. p. 22).

[Gen+16] Daniel Genkin, Lev Pachmanov, Itamar Pipman et Eran Tromer. "ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs". In : *Cryptographers' Track at the RSA Conference*. Springer. 2016, p. 219–235 (cf. p. 167).

[Gou15] Goubet, Lucien and Heydemann, Karine and Encrenaz, Emmanuelle and De Keulenaer, Ronald. "Efficient Design and Evaluation of Countermeasures against Fault Attack with Formal Verification". In : *14th Smart Card Research and Advanced Application Conference, CARDIS 2015*. Bochum, Germany, nov. 2015 (cf. p. 42–49, 51–53, 147).

[GS95] J. Güthoff et V. Sieh. "Combining software-implemented and simulation-based fault injection into a single fault injection method". In : *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. Juin 1995, p. 196–206. doi : 10.1109/FTCS.1995.466978 (cf. p. 151).

[GSM14] GSMA. *The Mobile Economy 2014*. http://www.gsmamobileeconomy.com/GSMA_ME_Report_2014_R2_WEB.pdf. En ligne ; accédé le 25 juillet 2016. 2014 (cf. p. 7).

[Har+12] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi et Pradeep Ramachandran. "Relyzer : exploiting application-level fault equivalence to analyze application resiliency to transient faults". In : *ACM SIGPLAN Notices*. T. 47. 4. ACM. 2012, p. 123–134 (cf. p. 152).

[Höl+15] Andrea Höller, Armin Krieg, Tobias Rauter, Johannes Iber et Christian Kreiner. "QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks". In : *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE. 2015, p. 530–533 (cf. p. 42, 43, 45–52, 106).

[IL09] Julien Iguchi-Cartigny et Jean-Louis Lanet. "Évaluation de l'injection de code malicieux dans une Java Card". In : *Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC'09)*. 2009 (cf. p. 22).

[Jen+94] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson et Johan Karlsson. "Fault injection into VHDL models : the MEFISTO tool". In : *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*. IEEE. 1994, p. 66–75 (cf. p. 38, 39).

[JT12] Marc Joye et Michael Tunstall. *Fault Analysis in Cryptography*. Springer, 2012 (cf. p. 36, 161).

[Kar+02] Ramesh Karri, Kaijie Wu, Piyush Mishra et Yongkook Kim. "Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers". In : *IEEE Transactions on computer-aided design of integrated circuits and systems* 21.12 (2002), p. 1509–1517 (cf. p. 161).

[Kar+09] Peter Karsmakers, Benedikt Gierlichs, Kristiaan Pelckmans, Katrien De Cock, Johan Suykens, Bart Preneel et Bart De Moor. *Side channel attacks on cryptographic devices as a classification problem*. Rapp. tech. COSIC technical report, 2009 (cf. p. v).

[Kau12] Xavier Kauffmann-Tourkestansky. "Analyses sécuritaires de code de carte à puce sous attaques physiques simulées". Thèse de doct. Université d'Orléans, 2012 (cf. p. 40).

[Kin76] James C. King. "Symbolic execution and program testing". In : *Communications of the ACM* 19.7 (1976), p. 385–394 (cf. p. 45).

[KJJ98] Paul C. Kocher, Joshua Jaffe et Benjamin Jun. *Introduction to Differential Power Analysis and Related Attacks*. Rapp. tech. En ligne ; dernière consultation le 23/12/2015. 1998 (cf. p. ii).

[KJJ99] Paul C. Kocher, Joshua Jaffe et Benjamin Jun. "Differential Power Analysis". In : *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. Springer-Verlag, 1999, p. 388–397. isbn : 3-540-66347-9. url : http://dl.acm.org/citation.cfm?id=646764.703989 (cf. p. ii, iv).

[KK99] Oliver Kömmerling et Markus G. Kuhn. "Design principles for tamper-resistant smartcard processors". In : *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*. WOST'99. Chicago, Illinois : USENIX Association, 1999. url : http://dl.acm.org/citation.cfm?id=1267115.1267117 (cf. p. v).

[KKA92] Ghani A. Kanawati, Nasser A. Kanawati et Jacob A. Abraham. "FERRARI : A tool for the validation of system dependability properties". In : *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. IEEE. 1992, p. 336–344 (cf. p. 37).

[Koc96] Paul C. Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In : *Advances in Cryptology—CRYPTO'96*. Springer. 1996, p. 104–113 (cf. p. ii).

[Kos+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham et al. "Experimental security analysis of a modern automobile". In : *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, p. 447–462 (cf. p. 167).

[KQ07] Chong Hee Kim et Jean-Jacques Quisquater. "Fault attacks for CRT based RSA : New attacks, new results, and new countermeasures". In : *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*. Springer, 2007, p. 215–228 (cf. p. 18, 23, 36, 144).

[KS01] Andrew Kennedy et Don Syme. "Design and implementation of generics for the .Net common language runtime". In : *ACM SigPlan Notices*. T. 36. 5. ACM. 2001, p. 1–12 (cf. p. 155).

[LA04] Chris Lattner et Vikram Adve. “LLVM : A compilation framework for lifelong program analysis & transformation”. In : *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, p. 75–86 (cf. p. 158).

[LB15] Julien Lancia et Guillaume Bouffard. “Java Card Virtual Machine Compromising from a Bytecode Verified Applet”. In : *14th Smart Card Research and Advanced Application Conference (CARDIS'15)*. Springer, 2015 (cf. p. 22).

[LDS78] Richard J Lipton, Richard A DeMillo et FG Sayward. “Hints on test data selection : Help for the practicing programmer”. In : *IEEE computer* 11.4 (1978), p. 34–41 (cf. p. 38).

[LE01] David Larochelle et David Evans. “Statically Detecting Likely Buffer Overflow Vulnerabilities”. In : *USENIX Security Symposium*. T. 32. Washington DC. 2001 (cf. p. 21).

[LH07] Daniel Larsson et Reiner Hähnle. “Symbolic Fault-Injection”. In : *International Verification Workshop (VERIFY)*. T. 259. 2007, p. 85–103 (cf. p. 38, 53).

[Lib13] Joint Interpretation Library. *Application of Attack Potential to Smartcards*. Rapp. tech. Version 2.9. Common Criteria, jan. 2013 (cf. p. 15, 17).

[Lin14] Lindeman, Todd. *“Smart” banking cards catching on everywhere but U.S.* https://www.washingtonpost.com/apps/g/page/business/smart-banking-cards-catching-on-everywhere-but-us/749/. En ligne ; accédé le 25 juillet 2016. Jan. 2014 (cf. p. 7).

[LR15] Benoit Laugier et Tiana Razafindralambo. “Misuse of Frame Creation to Exploit Stack Underflow Attacks on Java Card”. In : *14th Smart Card Research and Advanced Application Conference (CARDIS'15)*. Springer, 2015 (cf. p. 22).

[Mac+11] Jean-Baptiste Machemie, Clément Mazin, Jean-Louis Lanet et Julien Cartigny. “SmartCM a smart card fault injection simulator”. In : *IEEE International Workshop on Information Forensics and Security*. IEEE, 2011 (cf. p. 40, 42, 44–52).

[May87] Cathy May. “Mimic : A Fast System/370 Simulator”. In : *SIGPLAN Not.* 22.7 (juil. 1987), p. 1–13. issn : 0362-1340. doi : 10.1145/960114.29651. url : http://doi.acm.org/10.1145/960114.29651 (cf. p. 103).

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish et Robert H. Sloan. “Power analysis attacks of modular exponentiation in smartcards”. In : *Cryptographic Hardware and Embedded Systems*. Springer. 1999, p. 144–157 (cf. p. ii).

[Mil+97] Robin Milner, Mads Tofte, Robert Harper et David Macqueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. isbn : 0262631814 (cf. p. 135).

[Moo+02] Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins et George Taylor. “Improving Smart Card Security Using Self-Timed Circuits”. In : *Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems*. IEEE Computer Society. 2002, p. 211 (cf. p. 37).

[Moo+03] Simon Moore, Ross Anderson, Robert Mullins, George Taylor et Jacques J.A. Fournier. “Balanced self-checking asynchronous logic for smart card applications”. In : *Microprocessors and Microsystems* 27.9 (2003), p. 421–430 (cf. p. 37).

[Mor+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson et Emmanuelle Encrenaz. “Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller”. In : *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE. 2013, p. 77–88 (cf. p. 68).

[Mor14] Nicolas Moro. “Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués”. Thèse de doct. Université Pierre et Marie Curie-Paris VI, 2014 (cf. p. 47, 117).

[MP08] Wojciech Mostowski et Erik Poll. “Malicious code on Java Card smartcards : Attacks and countermeasures”. In : *Smart Card Research and Advanced Applications*. Springer, 2008, p. 1–16 (cf. p. 22).

[MSY06] Tal G. Malkin, François-Xavier Standaert et Moti Yung. “A comparative cost/security analysis of fault attack countermeasures”. In : *Fault Diagnosis and Tolerance in Cryptography*. Springer, 2006, p. 159–172 (cf. p. 161).

[Ord+15] Sébastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, Jean-Marc Dutertre et Philippe Maurine. “Evidence of a larger EM-induced fault model”. In : *CARDIS 2014, Smart Card Research and Advanced Application Conference*. T. 8968. Lecture Notes in Computer Science. Springer, 2015. isbn : 978-3-319-16762-6. doi : 10.1007/978-3-319-16763-3. url : http://dx.doi.org/10.1007/978-3-319-16763-3 (cf. p. 20, 79, ix).

[Pal05] Mike Pall. *LuaJIT*. http://luajit.org. 2005 (cf. p. 155).

[Pat+08] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk et Ravishankar Iyer. “SymPLFIED : Symbolic program-level fault injection and error detection framework”. In : *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE. 2008, p. 472–481 (cf. p. 38, 53).

[PC10] Ludovic Pièvre-Cambacédès et Claude Chaudet. “The SEMA referential framework : Avoiding ambiguities in the terms “security” and “safety””. In : *International Journal of Critical Infrastructure Protection* 3.2 (2010), p. 55–66 (cf. p. 35).

[Piè10] Ludovic Pièvre-Cambacédès. “Des relations entre sûreté et sécurité”. Thèse de doct. Télécom ParisTech, 2010 (cf. p. 35).

[Por+10] João Porto, Guido Araujo, Edson Borin et Youfeng Wu. “Trace execution automata in dynamic binary translation”. In : *International Symposium on Computer Architecture*. Springer. 2010, p. 99–116 (cf. p. 147).

[Pot+14] Marie-Laure Potet, Laurent Mounier, Maxime Puys et Louis Dureuil. “Lazart : A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections”. In : *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE. 2014, p. 213–222 (cf. p. 40, 42–44, 46–52, 117).

[QS02] Jean-Jacques Quisquater et David Samyde. “Eddy current for magnetic analysis with active sensor”. In : *Proceedings of Esmart*. T. 2002. 2002 (cf. p. 20).

[Res+03] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra et Nikil Dutt. “An efficient retargetable framework for instruction-set simulation”. In : *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*. IEEE. 2003, p. 13–18 (cf. p. 136).

[RG14a] Pablo Rauzy et Sylvain Guilley. “A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA”. English. In : *Journal of Cryptographic Engineering* 4.3 (2014), p. 173–185. ISSN : 2190-8508. DOI : 10.1007/s13389-013-0065-3. URL : http://dx.doi.org/10.1007/s13389-013-0065-3 (cf. p. 23, 41, 42, 44–49, 51, 52).

[RG14b] Pablo Rauzy et Sylvain Guilley. “Countermeasures Against High-Order Fault-Injection Attacks on CRT-RSA”. In : *FDTC'14 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, sept. 2014, p. 68–82. ISBN : 978-1-4799-6292-1. DOI : 10.1109/FDTC.2014.17 (cf. p. 23).

[Rig+04] Sandro Rigo, Guido Araujo, Marcus Bartholomeu et Rodolfo Azevedo. “ArchC : A SystemC-based architecture description language”. In : *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*. IEEE. 2004, p. 66–73 (cf. p. 133).

[Riv+14] Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne et Maxime Puys. “Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks”. In : *International Symposium on Foundations and Practice of Security*. Springer. 2014, p. 92–111 (cf. p. 159).

[Riv+15] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Bringer Julien et Laurent Sauvage. “High precision fault injections on the instruction cache of ARMv7-M architectures”. In : *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*. IEEE, 2015, p. 62–67. DOI : 10.1109/HST.2015.7140238. URL : http://dx.doi.org/10.1109/HST.2015.7140238 (cf. p. 68).

[Riv15] Lionel Rivière. "Securing software implementations against fault injection attacks on embedded systems". http://lionel.riviere.name/thesis/LionelRivierePhD.pdf. Thèse de doct. Paris : TELECOM ParisTech, sept. 2015 (cf. p. 73).

[Rot+04] Klaus Rothbart, Ulrich Neffe, Christian Steger, Reinhold Weiss, Edgar Rieger et Andreas Mühlberger. "High level fault injection for attack simulation in smart cards". In : *Test Symposium, 2004. 13th Asian.* IEEE. 2004, p. 118–121 (cf. p. 35, 39, 42, 43, 45–52).

[SA03] Sergei P. Skorobogatov et Ross J. Anderson. "Optical Fault Induction Attacks". In : *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems.* CHES '02. Springer-Verlag, 2003, p. 2–12. isbn : 3-540-00409-2. url : http://dl.acm.org/citation.cfm?id=648255.752727 (cf. p. 20).

[SBK10] Daniel Skarin, Raul Barbosa et Lohan Karlsson. "GOOFI-2 : A tool for experimental dependability assessment". In : *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on.* IEEE. 2010, p. 557–562 (cf. p. 38).

[SBN82] Daniel P. Siewiorek, Gordon Bell et Allen C. Newell. *Computer Structures : principles and examples.* McGraw-Hill, Inc., 1982 (cf. p. 35).

[Sch+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann et Olaf Spinczyk. "FAIL* : An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance". In : *Proceedings of the 11th European Dependable Computing Conference (EDCC'15).* 2015 (cf. p. 39, 53).

[SCS15] Horst Schirmeier, Borchert Christoph et Olaf Spinczyk. "Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors". In : *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15).* (Rio de Janeiro, Brazil). IEEE Computer Society Press, juin 2015, p. 319–330. doi : 10.1109/DSN.2015.44 (cf. p. 36, 119, 151).

[Seg+88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson et T. Lin. "FIAT-fault injection based automated testing environment". In : *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on.* Juin 1988, p. 102–107. doi : 10.1109/FTCS.1988.5306 (cf. p. 35, 37).

[SH07] Jörn-Marc Schmidt et Michael Hutter. "Optical and EM Fault-Attacks on CRT-based RSA : Concrete Results". In : *15th Austrian Workshop on Microelectronics(Austrochip 2007).* Verlag der Technischen Universität Graz, 2007, p. 61–67 (cf. p. 20).

[SIL11] Ahmadou Séré, Julien Iguchi-Cartigny et Jean-Louis Lanet. "Evaluation of Countermeasures Against Fault Attacks on Smart Cards". In : *International Journal of Security and Its Applications* 5.2 (2011), p. 49–61 (cf. p. 22).

[Sko05] Sergei P. Skorobogatov. *Semi-invasive attacks : a new approach to hardware security analysis*. Rapp. tech. This technical report is based on a dissertation submitted September 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College. University of Cambridge, Computer Laboratory, 2005 (cf. p. v, vi).

[Sko11] Sergei P. Skorobogatov. "Physical attacks on tamper resistance : progress and lessons". In : *Proc. of 2nd ARO Special Workshop on Hardware Assurance, Washington, DC*. 2011 (cf. p. vi).

[SKS12] Alexander Sepp, Julian Kranz et Axel Simon. "GDSL : a generic decoder specification language for interpreting machine language". In : *Electronic Notes in Theoretical Computer Science* 289 (2012), p. 53–64 (cf. p. 135).

[Smi+95] D. Todd Smith, Barry W. Johnson, Joseph A. Profeta et Daniele G. Bozzolo. "A method to determine equivalent fault classes for permanent and transient faults". In : *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*. Jan. 1995, p. 418–424. doi : 10.1109/RAMS.1995.513278 (cf. p. 151).

[ST04] Adi Shamir et Eran Tromer. "Acoustic cryptanalysis". In : *presentation available from http ://www. wisdom. weizmann. ac. il/ tromer* (2004) (cf. p. 167).

[STB97] Volkmar Sieh, Oliver Tschache et Frank Balbach. "VERIFY : evaluation of reliability using VHDL-models with embedded fault descriptions". In : *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. Juin 1997, p. 32–36. doi : 10.1109/FTCS.1997.614074 (cf. p. 38).

[SW79] Harry J. Saal et Zvi Weiss. "A Software High Performance APL Interpreter". In : *Proceedings of the International Conference on APL : Part 1*. APL '79. New York, New York, USA : ACM, 1979, p. 74–81. doi : 10.1145/800136.804441. url : http://doi.acm.org/10.1145/800136.804441 (cf. p. 103).

[TK10] Elena Trichina et Roman Korkikyan. "Multi fault laser attacks on protected CRT-RSA". In : *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE. 2010, p. 75–86 (cf. p. 23, 36).

[Voa96] Jeffrey Voas. "Testing software for characteristics other than correctness : Safety, failure tolerance and security". In : *Inproceedings of the 13th international conference on testing computer software*. Juin 1996 (cf. p. 18).

[Wag+13] Harry Wagstaff, Miles Gould, Björn Franke et Nigel Topham. "Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description". In : *Proceedings of the 50th Annual Design Automation Conference*. ACM. 2013, p. 21 (cf. p. 155).

[WMM04] Nicky Williams, Bruno Marre et Patricia Mouy. “On-the-fly generation of K-path tests for C functions”. In : *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society. 2004, p. 290–293 (cf. p. 45).

[WW05] Jason Waddle et David Wagner. “Fault attacks on dual-rail encoded systems”. In : *Computer Security Applications Conference, 21st Annual*. IEEE. 2005, 10–pp (cf. p. 37).

[WWM15] Mario Werner, Erich Wenger et Stefan Mangard. “Protecting the Control Flow of Embedded Processors against Fault Attacks”. In : *International Conference on Smart Card Research and Advanced Applications*. Springer. 2015, p. 161–176 (cf. p. 166).

Résumé. Lasers, impulsions électriques et électromagnétiques, confèrent à un attaquant le pouvoir mystérieux de perturber la logique de fonctionnement des appareils informatiques. Cette étonnante capacité peut s'avérer particulièrement néfaste pour les composants sécurisés, tels que les cartes à puce. Face à cette menace, la sécurité de ces composants est évaluée par des laboratoires spécialisés dans le cadre de normes internationales et sous l'égide d'agences gouvernementales. Cette thèse explore l'impact de l'injection de fautes, la conséquence des attaques par perturbation sur le code exécuté par le composant, dans le processus d'évaluation. On y élabore une nouvelle approche dite de bout en bout, pour lier les différentes parties du processus d'évaluation : l'analyse de code pour la détection de vulnérabilités et les attaques physiques sur carte. L'approche combine des modèles de fautes extraits durant les attaques physiques et un outil spécifiquement conçu pour extraire les vulnérabilités pertinentes et coter leur potentiel d'attaque. Enfin, on étudie l'impact sur la sécurité de l'injection de fautes multiples, qui multiplie le nombre de fautes injectables durant une seule exécution, et décuple ainsi le pouvoir de l'attaquant.

Mots-clés : Sécurité ; Vulnérabilités ; Modèle de faute ; Analyse dynamique ; Carte à puce ; Injection de fautes

Abstract. Lasers, electronics glitches and electromagnetic pulses, bestow upon an attacker the mysterious power to perturb the logic of operation of computing devices. This surprising ability may be especially harmful to secure hardware such as smartcards. Against this threat, the security of such hardware is assessed by dedicated laboratories according to international norms and under the auspices of national agencies. This thesis explores the impact of fault injection, which is the consequence of perturbation attacks on the code executed by a hardware device, in the evaluation process. We develop a novel end-to-end approach to close the gap between the analysis of the code for vulnerability detection and the physical attacks that are performed in the evaluation process. The approach combines fault models extracted during physical attacks with a specifically designed evaluation tool to extract relevant vulnerabilities and rate their attack potential. Lastly, we study the impact on security of multiple fault attack, a technique that significantly boosts the attacker's power by allowing several faults over the course of a single execution.

Keywords: Security; Vulnerabilities; Fault model; Dynamic analysis; Smartcard; Fault injection