Analyses sécuritaires de code de carte à puce sous attaques physiques simulées

Xavier Kauffmann-Tourkestansky

▶ **To cite this version:**

Xavier Kauffmann-Tourkestansky. Analyses sécuritaires de code de carte à puce sous attaques physiques simulées. Systèmes embarqués. Université d'Orléans, 2012. Français. NNT : . tel-00771273v1

HAL Id: tel-00771273

https://theses.hal.science/tel-00771273v1

Submitted on 8 Jan 2013 (v1), last revised 13 Apr 2013 (v4)

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

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

PUCVL
Pôle Universités
Centre Val de Loire

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

THÈSE présentée par :

Xavier KAUFFMANN-TOURKESTANSKY

soutenue le : **28 novembre 2012**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **INFORMATIQUE**

Analyses sécuritaires de code de carte à puce sous attaques physiques simulées

THÈSE dirigée par :

Pascal BERTHOMÉ	Professeur, ENSI de Bourges

RAPPORTEURS :

Samia BOUZEFRANE	Maître de conférences, CNAM
Nathalie DRACH-TEMAM	Professeur, UPMC Paris 6

JURY :

Mirian HALFELD FERRARI ALVES	Professeur, IUT d'Orléans, Président du jury
Jean-François LALANDE	Maître de Conférences, ENSI de Bourges
Jean-Louis LANET	Professeur, Université de Limoges
Francis CHAMBÉROT	Ingénieur, Oberthur Technologies

Remerciements

Je tiens à remercier les différentes personnes qui ont contribué à cette thèse.

Tout d'abord Emmanuel Prouff pour m'avoir donné la chance de faire une thèse chez Oberthur Card Systems (maintenant Oberthur Technologies). Ses conseils et remarques m'ont chaque fois guidé et m'ont poussé à améliorer la qualité de mon travail.

Je remercie Samia Bouzefrane ainsi que Nathalie Drach-Temam pour m'avoir fait l'honneur d'accepter d'être mes rapporteurs de thèse. Merci à elles et aux relecteurs attentifs qui ont su débusquer mes fautes et m'aider à les corriger.

Je tiens à remercier Francis Chambérot pour m'avoir accueilli dans son équipe de développeurs à Oberthur Technologies. Nos discussions ont grandement contribué à cette thèse et m'ont beaucoup appris sur le monde de l'entreprise.

Merci à tous mes collègues et amis qui ont rendu ces trois années riches en expériences et bons moments. Pour avoir répondu à toutes mes questions (pas toujours évidentes), nagé, bu et voyagé avec moi, vous avez fait de ces trois ans beaucoup plus qu'une simple thèse. Une dédicace spéciale au bureau 133, avec son ambiance du tonnerre, où pas une journée ne passe sans une bonne blague. Merci également aux habitants de la "zone crypto" qui n'ont jamais manqué de m'inviter à leurs pauses café et soirées arrosées. En tant que développeur, collègue et souvent ami, je garde d'excellents souvenirs de mon expérience dans votre équipe.

Merci aux thésards qui, à l'Ensi de Bourges, ont souffert avec LaTeXà mes côtés et m'ont permis d'attraper (parfois in extremis) mon train. Merci aussi aux autres personnes qui ont su, par leur importation de spiritueux ou leur poigne de fer, entretenir ma motivation pendant la rédaction de ce mémoire.

Je tiens également à remercier mes amis dont les randonnées et longues discussions ont renforcé mon physique et mon mental à intervalles réguliers.

Pour avoir donné matière à cette thèse et nos collaborations, je remercie tout particulièrement mes co-auteurs Karine Heydemann, Pascal Berthomé, Francis Chambérot et Jean-François Lalande.

Mes encadrants Pascal Berthomé et Jean François Lalande méritent une mention spéciale pour m'avoir si souvent encouragé, logé et nourri de leur passion pour l'informatique. Votre sens humain et votre travail ont beaucoup influencé le mien.

Je tiens aussi à remercier ma soeur Soline et Anh ma maman pour m'avoir soutenu et supporté tout au long de ces trois ans. Cette thèse vous est dédiée.

Mon dernier remerciement sera pour Agnès pour sa compréhension, son soutien et son amour qui m'ont beaucoup aidé dans l'écriture de ce manuscrit.

À mon père,

Table des matières

1 Introduction **1**

1.1 Cadre et enjeux . . . 1
1.1.1 Facteur d'échelle . . . 2
1.1.2 Déportement de la sécurité et environnement hostile . . . 3
1.1.3 Carte à puce et cryptographie . . . 4
1.1.4 Principe de fonctionnement d'une carte . . . 6
1.1.5 Diversité de l'écosystème . . . 7
1.1.6 Processus d'évaluation de la sécurité et de certification . . . 10
1.2 Approche fonctionnelle et vision bas niveau . . . 11
1.2.1 Fabrication et utilisation fonctionnelle . . . 12
1.2.2 Fonctionnement interne . . . 18
1.3 Besoins en sécurité . . . 27
1.3.1 Historique de la sécurité des cartes . . . 27
1.3.2 Attaques physiques . . . 28
1.3.3 Problématique globale . . . 30
1.3.4 Propriétés de sécurité . . . 31
1.3.5 Spécifications fonctionnelles et sécurités additionnelles . . . 32
1.3.6 Sécurisation et angles d'attaque . . . 38
1.3.7 Modélisation et vérification . . . 40
1.3.8 Angle d'approche et résumé . . . 44
1.4 Conclusion et annonce . . . 46

2 Modélisation et propriétés de sécurité **49**

2.1 Introduction, approche et hypothèses . . . 49
2.1.1 Approches existantes . . . 50
2.1.2 Décomposition de fonctionnalité . . . 53
2.1.3 Décomposition en éléments du langage . . . 53
2.2 Formalisme utilisé . . . 55
2.2.1 Information et conteneur . . . 55
2.2.2 Ensemble d'informations visibles . . . 56
2.2.3 Zone de garantie . . . 61
2.2.4 Type d'attaques . . . 61
2.2.5 Formalisation d'une propriété de sécurité . . . 61
2.3 Confidentialité d'une information . . . 62
2.3.1 Confidentialité sous attaque . . . 64
2.3.2 Zone de génération d'information d'un conteneur . . . 64
2.3.3 Zone de garantie de confidentialité . . . 66
2.3.4 Avantage pour l'attaquant et granularité de l'information . . . 67
2.4 Intégrité . . . 69
2.4.1 Accès à un conteneur et zone de garantie . . . 69

2.4.2 Intégrité en lecture et écriture . . . 70
2.4.3 Avantage pour l'attaquant . . . 73
2.4.4 Intégrité d'exécution . . . 73
2.5 Conclusion . . . 75
2.6 Perspectives . . . 77
2.6.1 Aspects sémantiques . . . 77
2.6.2 Temporalité des propriétés . . . 77
2.6.3 Prise en compte du compilateur . . . 77
2.6.4 Théorie de l'information . . . 78
2.6.5 Génération automatique de propriétés . . . 78

3 Modèle d'attaque 81
3.1 Introduction . . . 81
3.2 Caractérisation de l'attaquant . . . 82
3.2.1 Capacité d'observation . . . 82
3.2.2 Capacité d'action . . . 85
3.3 Modélisation des conséquences à haut niveau . . . 87
3.4 Modèle d'attaque physique à haut niveau . . . 88
3.4.1 Effets des attaques physiques . . . 88
3.4.2 Importance des attaques par NOP et par saut . . . 90
3.4.3 Attaques de code . . . 91
3.4.4 Attaques de données . . . 97
3.4.5 Décalage d'interprétation du code . . . 97
3.5 Conclusion . . . 104
3.6 Perspectives . . . 105
3.6.1 Prise en compte d'attaques d'un ordre supérieur . . . 105
3.6.2 Extension du modèle . . . 107

4 Vérification de sécurité sous attaques simulées 109
4.1 Introduction . . . 109
4.2 Vérification statique . . . 110
4.2.1 Exemple de propriétés de sécurité . . . 111
4.2.2 Vérification d'intervalle de valeurs . . . 113
4.2.3 Limites de la vérification et solution . . . 115
4.2.4 Instrumentation et génération d'annotations . . . 115
4.2.5 Vérification de sécurité et simulation d'attaques physiques . . . 118
4.3 Réduction du nombre d'attaques simulées . . . 118
4.3.1 Attaques considérées . . . 120
4.3.2 Typage de code . . . 120
4.3.3 Classes d'équivalence . . . 122
4.3.4 Conditions et boucles . . . 124
4.3.5 Expérimentations . . . 137
4.4 Conclusion . . . 138
4.5 Perspectives . . . 139

5 Test de sécurité et visualisation **141**
5.1 Introduction . . . 141
5.2 Méthodologie de test proposée . . . 143
5.2.1 Principe . . . 143
5.2.2 Méthode de classification des résultats . . . 146
5.3 Expérimentations et analyse sécuritaire . . . 148
5.3.1 Plate-forme expérimentale . . . 149
5.3.2 Exemple sur BZIP2 . . . 150
5.3.3 Exemple sur GZIP . . . 153
5.3.4 Couverture assembleur par le modèle C . . . 157
5.4 Analyse des résultats . . . 160
5.4.1 Distribution des attaques par taille du saut . . . 160
5.4.2 Analyse visuelle des fonctions vulnérables . . . 161
5.4.3 Implémentation de contre-mesures . . . 163
5.5 Résultats expérimentaux sur code de carte à puce . . . 164
5.6 Conclusion . . . 166
5.7 Perspectives . . . 167

6 Outils industriels **169**
6.1 Introduction . . . 169
6.2 Environnent et facteurs . . . 170
6.2.1 Environnement industriel . . . 170
6.2.2 Processus de vérification de la sécurité . . . 170
6.2.3 Facteurs à prendre en compte . . . 170
6.3 Outils et méthodes existantes . . . 171
6.3.1 Domaines similaires . . . 171
6.3.2 Cas particulier de la sécurité . . . 173
6.3.3 Techniques statiques et dynamiques . . . 173
6.3.4 Sécurité et absence d'erreurs . . . 173
6.4 Réalisations techniques . . . 175
6.4.1 Amélioration de la relecture de code . . . 175
6.4.2 Amélioration de la tracabilité fonctionnelle . . . 175
6.4.3 Vérification de formes de code . . . 176
6.4.4 Injection simple et exhaustive d'attaques . . . 177
6.4.5 Analyse et interrogation interactive du code source . . . 181
6.5 Conclusion . . . 186
6.6 Perspectives . . . 187

7 Conclusion **189**
7.1 Conclusion . . . 189
7.2 Perspectives . . . 190

Bibliographie **193**

Glossaire **207**

Table des Figures 211

Liste des Tableaux 212

Liste des Listings 213

Liste des Algorithmes 215

Chapitre 1

Introduction

Sommaire

1.1 Cadre et enjeux 1
1.1.1 Facteur d'échelle 2
1.1.2 Déportement de la sécurité et environnement hostile 3
1.1.3 Carte à puce et cryptographie 4
1.1.4 Principe de fonctionnement d'une carte 6
1.1.5 Diversité de l'écosystème 7
1.1.6 Processus d'évaluation de la sécurité et de certification 10
1.2 Approche fonctionnelle et vision bas niveau 11
1.2.1 Fabrication et utilisation fonctionnelle 12
1.2.2 Fonctionnement interne 18
1.3 Besoins en sécurité 27
1.3.1 Historique de la sécurité des cartes 27
1.3.2 Attaques physiques 28
1.3.3 Problématique globale 30
1.3.4 Propriétés de sécurité 31
1.3.5 Spécifications fonctionnelles et sécurités additionnelles 32
1.3.6 Sécurisation et angles d'attaque 38
1.3.7 Modélisation et vérification 40
1.3.8 Angle d'approche et résumé 44
1.4 Conclusion et annonce 46

1.1 Cadre et enjeux

Cette thèse s'intéresse aux attaques physiques contre les systèmes embarqués qui permettent de modifier le comportement du code. Provoquées par des vecteurs d'influences physiques extérieurs au système, ces attaques peuvent remettre en cause la sécurité de celui-ci. Cette thèse cible en particulier le code des cartes à puce. En effet, la sécurité du fonctionnement de ces cartes et la confidentialité des informations contenues dans celles-ci représentent un enjeu important.

Le cas des cartes à puce n'est cependant pas un cas isolé. En effet, on recense au cours des dernières années, divers cas de piratage où une vulnérabilité matérielle permet de compromettre l'ensemble d'un système en créant une faille logicielle exploitable. Un exemple très médiatisé est celui du hacker américain George Hotz alias "geohot" et ses "hacks" successifs de l'Iphone et de la Ps3 [Wikipedia 2012b]. Les schémas d'attaque utilisés se trouvent dans l'exploitation logicielle d'une mise en défaut du matériel par perturbation électrique [Hotz 2010]. Les conséquences financières de ces piratages sont estimées à plusieurs millions de dollars [Kushner 2012]. Dans le

milieu bancaire, ces schémas d'attaque sont connus [Rankl & Effing 2003] et pris très au sérieux parce qu'une remise en cause de la sécurité des cartes à puce entraînerait de graves conséquences financières ainsi qu'une perte de confiance de la part des utilisateurs. Si de telles attaques physiques sont connues, se prémunir contre elles reste un défi pour les personnes chargées de mettre en place la sécurité. L'auteur de [Leroy 2003] donne un aperçu de cette tâche de sécurisation et des difficultés associées en présentant quelques effets possibles d'attaques et les contre-mesures associées. Ce défi est non seulement technique, du fait de la frontière existant entre le monde du matériel et du logiciel qui entraine une difficulté à les modéliser simultanément. Ce défi est aussi industriel, à cause de contraintes fortes en termes de délais présents dans les projets réalisés en entreprise. Par conséquent, dans cette thèse, nous proposons d'étudier la sécurité logicielle de projets embarqués, tels que les cartes à puce, en incorporant la possibilité d'attaques physiques.

Avant de pouvoir être produites, ces cartes à puce doivent d'abord être certifiées. Cette certification a pour but de garantir leur résistance face à des attaques visant à compromettre la confidentialité ou l'intégrité des données mémorisées. Cette thèse s'inscrit également dans ce contexte de certification. D'un point de vue technique, nous nous intéressons à l'analyse de code source afin de :

- découvrir de nouvelles attaques ;
- mettre en place les contre-mesures appropriées.

Nous cherchons à fournir aux développeurs des techniques et outils pour mieux sécuriser leur code et garantir la sécurité des données sensibles manipulées.

L'enjeu principal de cette thèse, d'un point de vue industriel, est de permettre une meilleure sécurité des codes embarqués sur carte à puce par une validation rigoureuse de ces codes contre les attaques physiques. De plus, l'entreprise doit fournir un rapport aux organismes de certification tels que les *Critères Communs* mandatés par l'*Agence Nationale de la Sécurité des Systèmes d'Information* afin de garantir que des standards de sécurité sont bien respectés. Le respect de ces garanties permet d'assurer que des systèmes sensibles déployés à grande échelle, telles que les cartes à puce, sont capables de résister à des attaques malicieuses.

D'un point de vue académique, le thème de cette thèse se trouve entre deux domaines : l'électronique embarquée et l'informatique. L'enjeu est de créer des ponts entre ces deux domaines au travers d'une modélisation des effets d'attaques matérielles sur le code source du programme. Le but de ces ponts est d'allier ces deux domaines afin d'obtenir une meilleure sécurité globale. L'originalité du travail se trouve dans le point de vue adopté qui apporte une approche orientée logicielle à une modélisation physique des attaques. Ces travaux peuvent servir de point de départ à de nouveaux axes d'approche au problème de la sécurité dans le domaine embarqué. Ils peuvent aussi permettre une intégration de nouvelles analyses à des systèmes logiciels de vérification existants qui n'embarquent pas cette dimension de fautes matérielles. Avant de parler en détail du travail effectué, nous présentons quelques aspects liés à la sécurité des systèmes embarqués et en particulier aux cartes à puce.

1.1.1 Facteur d'échelle

Le contexte d'utilisation particulier des cartes à puce entraine des conséquences sur la sécurité de celles-ci et des considérations liées à leur diffusion à grande échelle. Leur utilisation dans un environnement non sécurisé physiquement pose aussi des problèmes de sécurité.

Marché, environnement et perspectives Le rapport [Consultants 2009] donne un bilan de la situation actuelle sur le marché mondial de la carte à puce. Des perspectives d'évolution et des scénarii stratégiques sont aussi présentés. Ce rapport montre l'enjeu stratégique et économique que représente la carte à puce au niveau mondial. La migration vers ce support sécurisé a sensiblement contribué, au cours des 20 dernières années, à réduire la fraude bancaire liée au paiement. Les auteurs estiment qu'avec l'émergence des technologies sans contact, les cartes à puce resteront dans les années à venir un des supports sécurisés de référence porté notamment par les industries de télécommunication. En effet, les cartes à puce sont un moyen simple et à bas coût permettant d'intégrer une composante garantissant un haut niveau de sécurité à des schémas qui nécessitent le partage d'une information (par exemple un schéma d'authentification).

Nombre de cartes en circulation : effet de masse amplificateur pour la sécurité La carte à puce est ainsi un médium d'authentification largement déployé et utilisé. D'après l'auteur de [Avenel 2010], le nombre de cartes à puce livrées dans le monde a été multiplié par 10 entre 2000 et 2010, passant de 540 millions à près de 5 milliards. Selon [Bodescot & Guinot 2011], l'année 2011 s'achève avec "une hausse de 11% des ventes mondiales de cartes à microprocesseur (carte bancaire, téléphone avec carte SIM), à plus de 6 milliards d'unités". Les prévisions mentionnées dans [Avenel 2010] annoncent que le nombre de cartes livrées dans le monde pourrait être multiplié par 4 au cours des 10 prochaines années pour atteindre 20 milliards. Ce déploiement de masse amplifie les risques associés à la sécurité et nécessite de la part des fabricants de garder une longueur d'avance sur les capacités des attaquants. Une grande réactivité en cas de découverte de nouvelles attaques est aussi requise de leur part.

1.1.2 Déportement de la sécurité et environnement hostile

Le but d'une carte à puce est d'assurer un certain nombre de fonctionnalités. Dans le cas d'une carte bancaire, ces fonctionnalités consistent à permettre à un utilisateur d'effectuer des transactions et de retirer de l'argent. Ces opérations nécessitent plusieurs acteurs, notamment la banque et l'utilisateur. Le rôle d'une carte est d'authentifier l'utilisateur auprès de la banque afin que celle-ci valide ou non la transaction bancaire demandée.

Dans ce schéma d'authentification, certaines informations partagées entre les différents acteurs sont déportées à l'intérieur de la carte remise au porteur. Cette carte et les informations qu'elle contient représente un des éléments de sécurité du schéma. Associée à une information connue seulement du porteur et le différenciant d'un autre, une authentification du porteur et de la carte est réalisée permettant un retrait ou une transaction. La carte est donc un élément du processus d'*authentification à deux facteurs* pour le porteur (une information qu'il connait et un élément physique qu'il possède).

Cependant, pour remplir son rôle dans une transaction, une carte doit disposer d'informations permettant de l'identifier, ainsi que son porteur, auprès de la banque. Les cartes renferment donc des secrets appartenant à la banque et par conséquent n'appartiennent pas à l'utilisateur final. De plus, afin de remplir leur rôle de jeton d'authentification, ces cartes doivent être en libre circulation et par conséquent des personnes malveillantes peuvent facilement se les procurer. En fait, une fois la carte sortie de l'environnement contrôlé d'une usine, elle se trouve dans un environnement où chacun des acteurs peut être considéré comme hostile. Que ce soit un utilisateur ou un commerçant malveillant ou un attaquant quelconque, tous peuvent avoir

un intérêt à exposer des secrets contenus dans la carte ou à modifier son comportement. Par conséquent, chacun d'eux peut mettre en danger la sécurité de la carte.

La sécurité dans ces systèmes d'authentification repose sur des informations sensibles contenues dans un objet physique qu'une tierce personne n'est pas en mesure de compromettre. Le cas échéant, cette tierce personne sera en mesure de se faire passer pour le porteur original auprès de la banque et aura accès aux ressources de celui-ci. Différents autres scénarii d'attaques visant des gains différents sont envisageables. Certains de ces scénarii sont évoqués dans la section 1.1.6.

1.1.3 Carte à puce et cryptographie

L'un des principaux avantages des cartes à puce est de pouvoir embarquer à bas coût et à encombrement réduit des capacités de calcul élevées permettant des calculs mathématiques sophistiqués. Ces calculs participent à la sécurité du système en offrant la possibilité d'intégrer des opérations cryptographiques de chiffrement et de signature à des schémas d'authentification. La cryptographie tient ainsi un rôle important dans le fonctionnement des cartes à puce. Elle assure notamment la confidentialité des données enregistrées dans la carte mais surtout la confidentialité des messages transmis entre un émetteur (le porteur) et un récepteur (la banque). Un schéma de chiffrement utilise une clé afin de transformer le message à transmettre en un message chiffré. Ce message peut seulement être déchiffré par un destinataire qui possède la clé de déchiffrement adéquate. On distingue les chiffrements :

symétrique ou à clé privée (par exemple le DES, Triple DES et l'AES) qui utilise une clé commune pour le chiffrement et le déchiffrement. Cette clé est partagée entre l'émetteur et le destinataire et doit être communiquée par un des correspondant au second correspondant. Si un attaquant intercepte cette clé pendant cette communication, il sera en mesure de déchiffrer les messages transmis entre les deux correspondants. La sécurité de ce type de chiffrement repose sur la non divulgation d'un secret commun entre les deux correspondants.

asymétrique ou à clé publique (par exemple le RSA ou les ECC voir ECC) qui utilise une paire de clés dont l'une peut être rendue publique. Ce type de chiffrement renforce la sécurité car la clé privée n'a pas besoin d'être communiquée entre les correspondants. Un attaquant, s'il connaît la clé publique et est capable d'intercepter les messages chiffrés entre les correspondants, n'est pas en mesure de déchiffrer ces messages. Cette résistance est due à la difficulté à factoriser les grands nombres entiers. La sécurité de ce type de chiffrement repose essentiellement sur la non divulgation de la clé privée.

à apport nul de connaissance (zero knowledge) qui repose sur un système de stimulation/réponse par un des correspondants pour amener l'autre correspondant à fournir la preuve de la connaissance d'une information [Goldwasser *et al.* 1989]. Cette méthode permet d'authentifier un des correspondants sans communiquer d'informations sur le secret utilisé à un éventuel attaquant [Quisquater *et al.* 1989].

Pour un échange de messages chiffrés entre une carte et une banque, un algorithme à clé publique nécessite deux paires de clés. La carte abrite sa clé privée qui va servir à déchiffrer les messages chiffrés par la banque avec la clé publique correspondante. La carte possède aussi la clé publique de la banque qu'elle utilisera pour chiffrer des messages à destination de la banque.

Un chiffrement asymétrique peut aussi être utilisé dans un schéma de signature où l'auteur d'un message veut prouver qu'il est bien à l'origine du message transmis. Dans cette configuration, la carte peut générer une paire de clés asymétriques. Le message sera signé à l'aide de la clé privée puis transmis avec la clé publique. Un destinataire recevant le message pourra ainsi vérifier que la carte est bien à l'origine du message transmis en validant celui-ci à l'aide de la clé publique.

Les algorithmes de chiffrement requièrent, dans certains cas, (par exemple, pour le RSA) des modules matériels additionnels capables de réaliser des multiplications et des exponentiations. La plupart des cartes à puce qui utilisent des algorithmes à clés publiques incorporent un cryptoprocesseur dédié. Une carte peut aussi embarquer un module dédié à la génération d'aléa. En effet, certains protocoles d'authentification ou des opérations cryptographiques comme la génération de clés confidentielles ou de vecteurs d'initialisation nécessitent la génération de nombres aléatoires. La génération de nombres réellement aléatoires est importante en cryptographie. Parmi ces modules dédiés à la génération d'aléa, les Physical Unclonable Functions ou PUF sur silicium sont des primitives cryptographiques récentes qui utilisent les différences physiques entre des circuits d'architecture identique pour générer un code spécifique à chaque circuit (semblable à une empreinte digitale unique du circuit).

Le risque au niveau de la génération de nombres aléatoires apparaît quand, par manque de diversification d'un calcul, une même valeur est constamment obtenue en sortie ou quand des collisions apparaissent dans ce calcul (plusieurs entrées donnant la même sortie). Il serait dans ce cas possible de remonter jusqu'à un secret utilisé dans le calcul. Si le nombre aléatoire est utilisé pour augmenter l'entropie d'une valeur, il sera possible d'obtenir des valeurs constantes. Si cette constante est utilisée en entrée d'un calcul, il serait alors encore une fois possible de remonter jusqu'à un secret utilisé dans ce calcul.

Le choix d'un algorithme cryptographique dépend surtout du but fonctionnel recherché et du schéma dans lequel la carte à puce est utilisée. Le choix d'un algorithme particulier conditionnera ainsi certaines caractéristiques matérielles du composant.

Indépendamment du besoin fonctionnel, la cryptographie est généralement utilisée dans les applications embarquées au niveau de :

- l'authentification de la carte (exemple carte SIM) ;
- l'authentification du porteur (code PIN) ;
- la génération et vérification de signatures électroniques : pour le transfert de fichiers, les certificats ...

Ces parties sensibles sont les cibles d'attaques visant à compromettre les éléments sensibles manipulés. Des travaux ont été menés, notamment sur les attaques par canaux cachés qui permettent, par simple observation de la consommation électrique ou du rayonnement électromagnétique de la carte, de déduire des informations sur les données et secrets manipulés. Nous aborderons la capacité d'observation d'un attaquant dans le chapitre 3.2.1.

Les implémentations cryptographiques sont aussi vulnérables aux attaques par fautes qui, en introduisant une erreur dans l'algorithme cryptographique, peuvent compromettre celui-ci et permettre à l'attaquant de remonter aux clés utilisées. La capacité d'action d'un attaquant sera abordée dans le chapitre 3.2.2.

Le terme d'attaque par "canaux cachés" ou "canaux auxiliaires" est utilisé pour qualifier ces attaques physiques sur un système. Ces attaques exploitent des vecteurs physiques permettant d'observer ou d'influencer le comportement du composant lors de son fonctionnement.

Dans cette thèse, nous n'abordons pas en détail les attaques par canaux cachés sur les parties cryptographiques. Les parties cryptographiques ne représentent qu'une partie du système exposée aux attaques physiques. De plus, la sécurité de l'ensemble d'un système repose sur d'autres éléments de sécurité qui sont aussi vulnérables à ces attaques, en particulier aux attaques par fautes. Nous cherchons donc, dans la suite de cette thèse, à caractériser les effets des attaques par canaux cachés et à montrer comment le code d'un projet embarqué peut être sensible à de telles attaques.

Pour conclure sur l'aspect cryptographique, on peut mentionner des domaines proches comme l'obfuscation de code ou la cryptographie en boîte blanche [Joye 2008] qui sont aussi utilisés afin d'assurer la confidentialité de données sensibles.

1.1.4 Principe de fonctionnement d'une carte

Prenons comme exemple un schéma de paiement par carte bleue. Celui-ci peut se décomposer en 3 parties :

Authentification de la carte qui se fait hors-ligne (sans appeler la banque). Des informations relatives au porteur (nom, numéro de carte, date de validité...) ainsi qu'une valeur de signature VS sont inscrites dans la carte. La signature VS est calculée de manière définitive à la personnalisation. Y, une valeur numérique déduite des informations écrites dans la carte (par une fonction de hachage), est d'abord calculée. Notons $Y = f(info)$. VS est alors calculée en utilisant la clé secrète S du groupement des cartes bancaires (GIE CB) : $VS = S(Y)$. La fabrication et l'écriture des VS sur la puce se fait dans des locaux très sécurisés, car S doit rester secret. Lorsque la carte est introduite dans le terminal, celui-ci lit les informations portées par la carte et la valeur de signature VS. Il calcule alors $Y1 = f(info)$ et $Y2 = P(VS) = P(S(Y))$, P étant la clé publique du GIE. Puis il compare $Y1$ et $Y2$: pour qu'une carte soit valide, il faut que $Y1 = Y2$. Ces fonctions à clé secrète et à clé publique sont basées sur le RSA. Le modulo publique français est un nombre connu entre 768 et 1024 bits, produit de 2 premiers inconnus.

Authentification du porteur de la carte avec un secret où un secret est chiffré et stocké dans la puce de la carte. C'est la puce qui vérifie elle-même si le code présenté par le porteur est le bon. Elle transmettra ensuite un jeton d'authentification au terminal prouvant que l'authentification a réussi.

Authentification en ligne qui n'est pas réalisée pour toutes les transactions, mais seulement pour celles dépassant un certain montant. Le terminal interroge la banque, qui envoie à la carte une valeur aléatoire x. La carte calcule $y = f(x, K)$, où K est une clé secrète inscrite dans la partie illisible de la carte et f la fonction de chiffrement du DES (ou du triple DES depuis 1999). La valeur est transmise à la banque, qui calcule elle-même $f(x, K)$ et donne ou non l'autorisation. Remarquons que ceci nécessite que la banque connaisse la clé secrète de toutes les cartes.

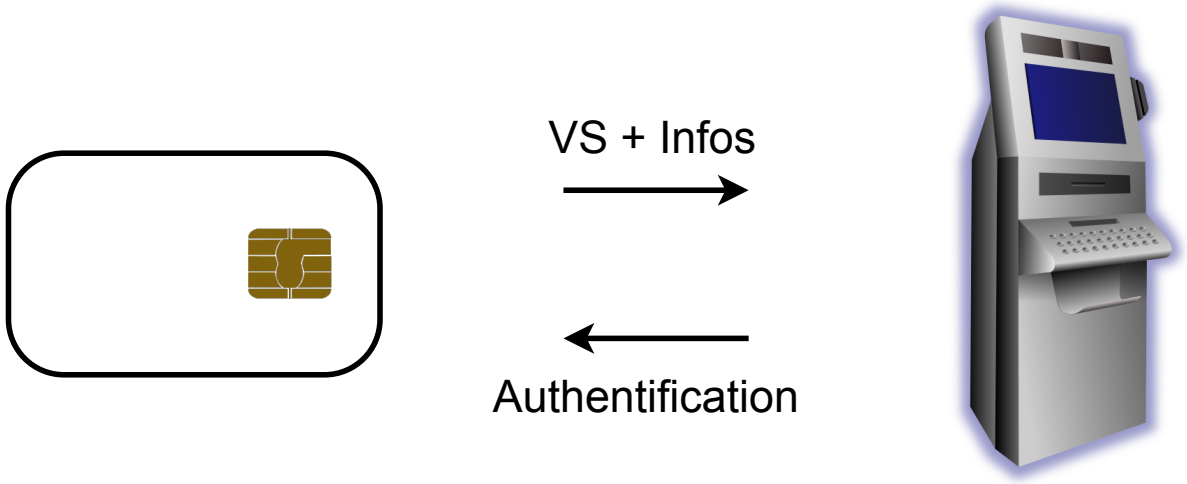


FIGURE 1.1 – Schéma d'une authentification hors ligne ne nécessitant pas de participation de la banque dans la transaction [Bayart 2012]

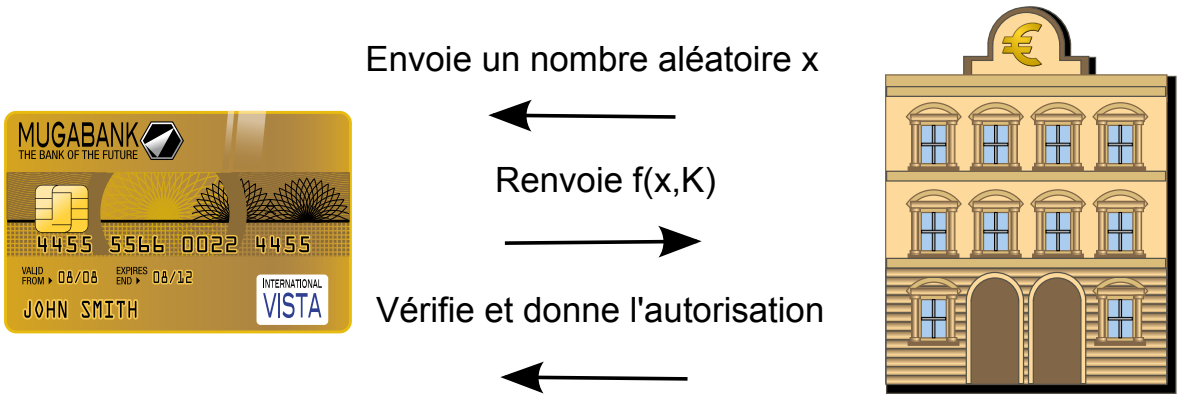


FIGURE 1.2 – Schéma d'une authentification en ligne authentifiant la carte auprès de la banque au moment de la transaction [Bayart 2012]

La figure 1.1 illustre le mécanisme d'authentification de la carte dans un scénario ne nécessitant pas d'authentification auprès de la banque (typiquement pour des faibles montants). La figure 1.2 illustre le mécanisme d'authentification de la carte dans un scénario nécessitant une authentification auprès de la banque.

1.1.5 Diversité de l'écosystème

Nous avons mentionné l'intérêt principal de la carte à puce qui est de pouvoir embarquer sur une plate-forme électronique des capacités de calcul élevées permettant des opérations cryptographiques qui assurent la sécurité de données confidentielles. Ces plates-formes de taille et

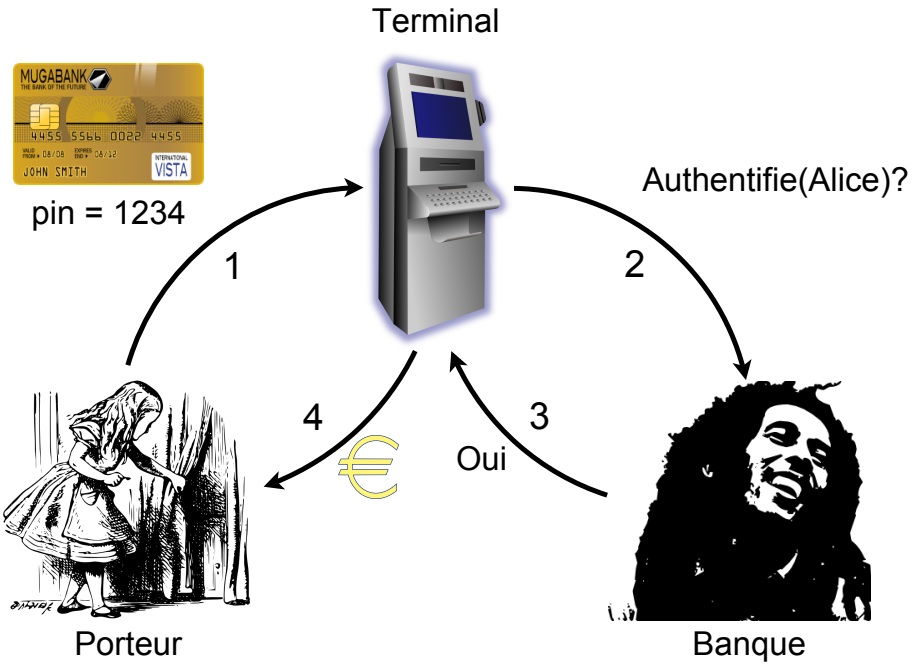


FIGURE 1.3 – Schéma d'authentification bancaire

consommation réduites et possédant un prix peu élevé, sont incorporées dans différents schémas d'authentification. Les cartes à puce sont ainsi utilisées dans de nombreux secteurs dans lequel l'écosystème nécessite que l'utilisateur final dispose d'un moyen de prouver sa légitimité à un autre acteur du système [Descamps 2000].

Le contrôle d'accès dans lequel le porteur de la carte prouve par sa possession physique son appartenance à un groupe habilité à accéder à un environnement restreint ;

Les cartes d'identité qui permettent par leur possession physique et la correspondance des informations contenues dans celles-ci avec les caractéristiques morphologiques du porteur d'authentifier celui-ci ;

La téléphonie mobile dans laquelle la puce identifiera l'appareil et implicitement l'utilisateur auprès de son fournisseur d'accès afin que celui-ci l'autorise à accéder à son réseau ;

La télévision numérique dans laquelle la carte va permettre au consommateur final équipé d'un décodeur de déchiffrer le contenu multimédia envoyé par le groupe audiovisuel afin de le visualiser ;

La carte multi-services pouvant regrouper des offres de fidélisation et d'autres applications comme par exemple des compteurs d'utilisation ;

Les cartes de paiement bancaire permettant au porteur de la carte, à condition de connaitre le PIN associé, de s'authentifier auprès de la banque et de réaliser des transactions.

Les deux schémas 1.3 et 1.4 illustrent l'utilisation d'une carte à puce dans des cas courants où elle intervient dans le processus d'authentification de l'utilisateur avec un autre correspondant.

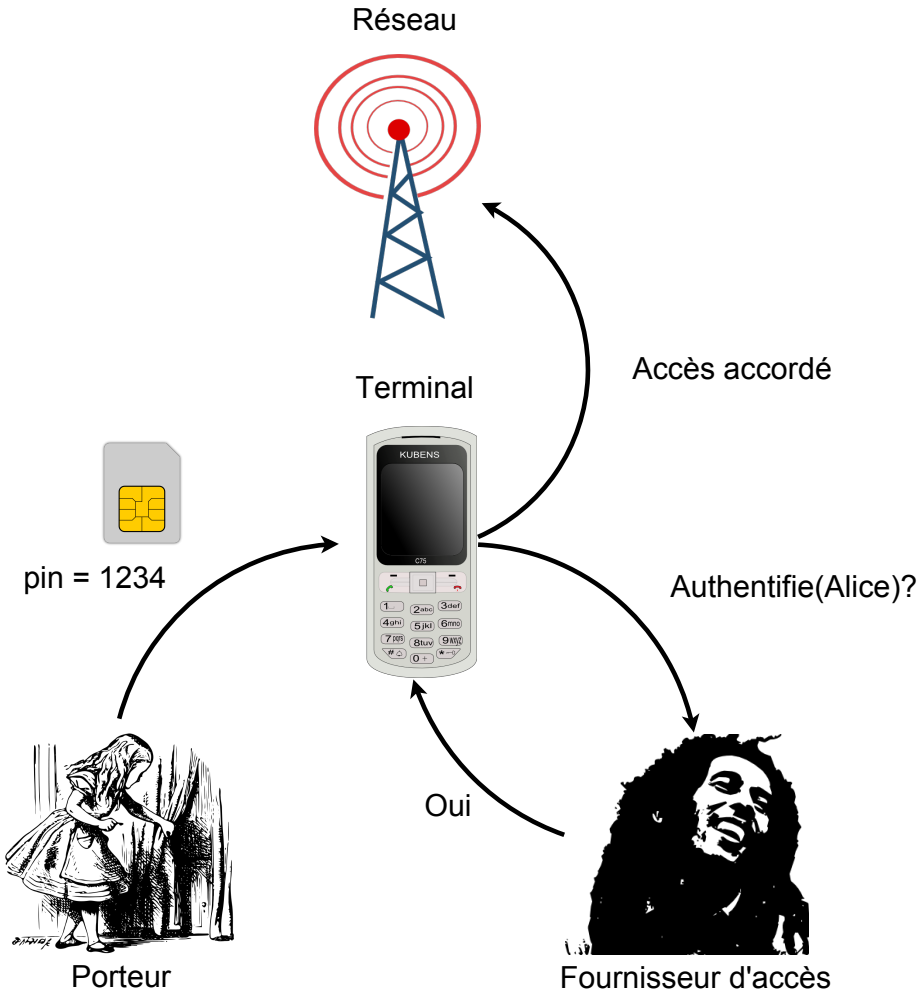


FIGURE 1.4 – Schéma d'authentification téléphonique

Le rôle d'une carte à microprocesseur peut aller du simple service d'authentification jusqu'à servir de plate-forme de support pour des applications tierces comme pour les Javacards. Des services sont connus du grand public comme les cartes Mifare d'authentification, les Passe Navigo pour le transport, les cartes SIM pour la téléphonie, les cartes Canal+ pour la télévision, les cartes VISA / Mastercard et les portefeuilles électroniques Moneo pour le domaine bancaire. Ces exemples montrent l'intégration de cette technologie dans la vie courante et la variété de l'écosystème lié à la carte à puce.

Au cœur d'une carte à puce, comme pour tout système embarqué, se trouve le microcontrôleur. Parmi les différentes familles de microcontrôleur utilisées en embarqué, on peut citer le 8051 et ses variantes sur lequel est basé un bon nombre de cartes bancaires du marché appelées. Les cartes mobiles et d'identité, qui demandent des fonctionnalités d'interopérabilité et une versatilité post production plus grande, sont plus souvent basées sur des plates-formes Java. Ces plates-formes respectent les spécifications Java Card qui définissent un standard pour toutes les cartes de ce type. Parmi ces plates-formes, on retrouve des plates-formes multi-applicatives telles

que OpenCard ou MultOS qui disposent déjà d'un système d'exploitation. Ces cartes offrent aux développeurs des interfaces de haut niveau rendant l'accès aux services de bas niveau transparent. Cette abstraction facilite l'implémentation rapide d'applications. L'existence de telles cartes nous montre qu'il est difficile de développer une application à un niveau fonctionnel tout en gérant les contraintes de bas niveau. Ces cartes nous montrent aussi que certaines entreprises développent spécifiquement des applications. Ces entreprises développeront ainsi une compétence uniquement fonctionnelle et délégueront la sécurité de bas niveau à d'autres spécialistes.

Pour plus de détails sur les différents cas d'utilisation d'une carte à puce ainsi que de nombreuses autres informations reliées au sujet, le lecteur pourra se référer à [Bouzefrane 2009].

Cette diversité et généralisation (tout comme le facteur d'échelle mentionné dans la section 1.1.1) représentent souvent des facteurs aggravants du point de vue de la sécurité et nécessite le recours à des techniques de sécurisation spécifiques et adaptées. En effet, si compromettre la sécurité d'une carte revenait à compromettre l'ensemble des cartes du même type présentes en circulation, le danger pour le système serait bien trop élevé. Pour combler cette vulnérabilité, on utilise des techniques comme la *diversification*, permettant, en se basant sur un identifiant unique comme le numéro de série de la carte ou un nombre généré aléatoirement, de différencier des opérations sensibles d'une carte à une autre.

La diversité des cas d'utilisation mentionnés précédemment nécessitent donc de prendre en compte différemment la sécurité dans les différents scénarii où une carte à puce est utilisée. Cependant une procédure d'évaluation commune est nécessaire afin de certifier le respect d'un niveau de sécurité des cartes produites et aussi standardiser les attaques prises en compte dans cette évaluation.

1.1.6 Processus d'évaluation de la sécurité et de certification

Une fois créées et avant de pouvoir être vendues, les cartes doivent être certifiées par une autorité de certification (en France, l'ANSSI). Cette autorité va mandater des centres spécialisés, les CESTI, chargés d'effectuer des tests de vulnérabilité afin de confirmer que certains schémas de sécurité sont bien respectés sur ces cartes au moment de leur création et leur développement.

Le document [Criteria 2009] propose des exemples d'attaques (duplication de la carte, découverte des secrets internes, modification de fonctionnement), des exemples d'attaquants. Il propose aussi une classification et notation des attaques en fonction de critères comme le nombre de cartes détruites avant que l'attaque n'aboutisse, le temps nécessaire pour réaliser l'attaque ou les compétences de l'attaquant. Ce document classe les attaques par critères de faisabilité et standardise une méthode afin d'évaluer le risque que représente une attaque sur un composant matériel. [Eurosmart 2001] est un document similaire qui décrit aussi bien les attaques potentielles qu'une méthode d'évaluation des cartes contre les attaques décrites. Ces documents, réalisés par de nombreux spécialistes en sécurité dans le domaine embarqué, montrent que les attaques physiques sont des dangers bien réels et sont envisagées très sérieusement surtout dans le milieu bancaire. Ce document fait partie d'un ensemble de documents décrivant une méthodologie d'évaluation de la sécurité regroupé sous l'appellation "Critères Communs".

La décision d'intégrer les Critères Communs comme processus d'évaluation et de faire appel à des centres d'évaluation indépendants ont fait suite à diverses fraudes à grande échelle. Parmi celles-ci, l'affaire "Humpich" et sa "Yes Card" ont forcé le gouvernement français à réévaluer les processus de certification au niveau national. Serge Humpich avait réussi à contourner deux

systèmes de sécurité existants : premièrement il avait réussi à créer des cartes qui, quel que soit le code (authentifiant le porteur) présenté, renvoyait toujours "code bon" ; deuxièmement, il avait contourné l'authentification hors ligne RSA. Comme mentionné précédemment en section 1.1.3, cette sécurité repose sur la difficulté à factoriser les grands nombres entiers. Or, en 1998, l'entier n utilisé par le GIE avait une taille de 320 bits (inchangée depuis 1990). Cependant, il était possible à cette époque de factoriser des entiers jusqu'à 512 bits. Humpich en utilisant un logiciel de factorisation a ainsi réussi à factoriser cet entier et à découvrir la clé secrète. Depuis la taille est passée à 768 bits. Similairement, l'authentification en ligne est passée du simple DES au Triple DES lui aussi victime de l'augmentation des capacités de calcul.

En parallèle des évaluations de sécurité menées dans le cadre des Critères Communs, des groupements bancaires, comme le GIE CB, évaluent de leur côté la résistance des produits délivrés face aux attaques physiques. Pour des hauts niveaux de sécurité, les processus d'évaluation demandent la preuve formelle de garanties de sécurité telles que la confidentialité, l'intégrité ou le cloisonnent de données. Ces preuves sont demandées afin de pouvoir atteindre des niveaux de certification de sécurité recommandés pour l'utilisation de cartes comme les cartes bancaires ou les cartes passeports électroniques.

Pour plus d'information sur les Critères Communs, les processus d'évaluation associés et leur mise en application, le lecteur peut se référer à [CCP 2012]. Le lecteur peut aussi se référer à [Flottes *et al.* 2011] qui liste les principaux acteurs dans le domaine de la sécurité embarquée ainsi que les défis existants dans le domaine et les axes de recherche actuels.

Les processus d'évaluation et la recherche autour des attaques physiques confirment le danger qu'elles représentent dans des secteurs aussi sensibles que le secteur bancaire. Si on décompose une carte à puce comme un système logiciel s'exécutant sur une plate-forme matérielle, on peut admettre que pour réussir à sécuriser convenablement une carte, il convient de comprendre aussi bien son fonctionnement logiciel que matériel ainsi que les moyens dont dispose un attaquant à chacun de ces niveaux pour compromettre la sécurité.

Si par une attaque, il est possible de violer la confidentialité d'une information, compromettre l'intégrité d'une donnée ou fausser la logique d'exécution, la sécurité de l'ensemble du système peut être mise en défaut. Il est donc important de s'intéresser aux besoins en sécurité de ce système et aussi à leur garantie face aux attaques. Afin de cerner ces besoins en sécurité, un formalisme les décrivant est nécessaire. Dans le chapitre 2, nous proposerons un formalisme pour des propriétés de sécurité telles que la confidentialité et l'intégrité de données ou l'intégrité d'exécution.

Les outils développés dans le chapitre 6 visent à vérifier les propriétés de sécurité formalisées. Ils peuvent aussi être utilisés dans une phase de validation de la sécurité.

1.2 Approche fonctionnelle et vision bas niveau

Dans la section 1.2.1, nous nous intéressons à la fabrication des cartes à puce et plus particulièrement au processus de développement logiciel. Nous plaçons le processus de certification de sécurité en perspective au sein de celui-ci. Nous parlons ensuite des différents types de cartes, leur communication avec un terminal et illustrons différents cas d'utilisation. Dans un deuxième temps, nous donnons, dans la section 1.2.2, des détails techniques sur le fonctionnement interne d'un microcontrôleur afin de détailler l'architecture de celui-ci. Cette étape permet de mieux

comprendre l'impact des attaques physiques et leurs conséquences sur le code source.

1.2.1 Fabrication et utilisation fonctionnelle

Processus de création général

Le processus de création général d'une carte suit 6 étapes menant à son utilisation par le porteur final. Le rôle joué par les différents acteurs responsables de sa création, leur compétence propre et la chronologie de la création sont visibles au travers de ces étapes.

- Etape 1 : la conception et fabrication de la puce ;
- Etape 2 : le développement du logiciel embarqué ;
- Etape 3 : le masquage du logiciel dans la puce ;
- Etape 4 : l'assemblage de la carte à puce ;
- Etape 5 : la personnalisation ;
- Etape 6 : l'utilisation.

Les concepteurs de la partie logicielle et la partie matérielle sont souvent deux entreprises différentes. L'entreprise qui développe l'applicatif achètera une puce à un fondeur suivant des spécifications propres au produit final demandé par le client. Le produit final, la puce embarquant le logiciel répondant aux besoins du client (par exemple une banque), sera ensuite livré.

Ainsi, d'un point de vue chronologique, la conception et la fabrication de la puce ont déjà été réalisées par le fondeur, qui propose des composants au concepteur logiciel. Celui-ci choisit le plus approprié par rapport au projet que souhaite le client final. Une fois la partie logicielle achevée, un premier "masquage" du logiciel sur le composant est effectué avant une production en masse. L'assemblage de la puce avec le support plastique est réalisée. Une production de masse est ensuite lancée où une phase de personnalisation configure les cartes avec des informations propres à l'utilisation finale. Les cartes sont finalement distribuées aux utilisateurs.

Processus de création logiciel

Dans cette thèse, nous nous intéressons à la sécurité logicielle d'un système embarqué soumis à des attaques physiques. Par conséquent, analyser le processus de développement logiciel est nécessaire. Cette analyse permet notamment d'appréhender les contraintes associées au développement en terme de temps et de moyens ainsi que les contraintes d'intégration pour une méthodologie de vérification de la sécurité.

La figure 1.5 illustre le processus de développement du logiciel embarqué dans une carte à puce. Dans ce processus, il existe deux étapes de vérification de la sécurité qui interviennent d'abord au niveau logiciel puis au niveau matériel.

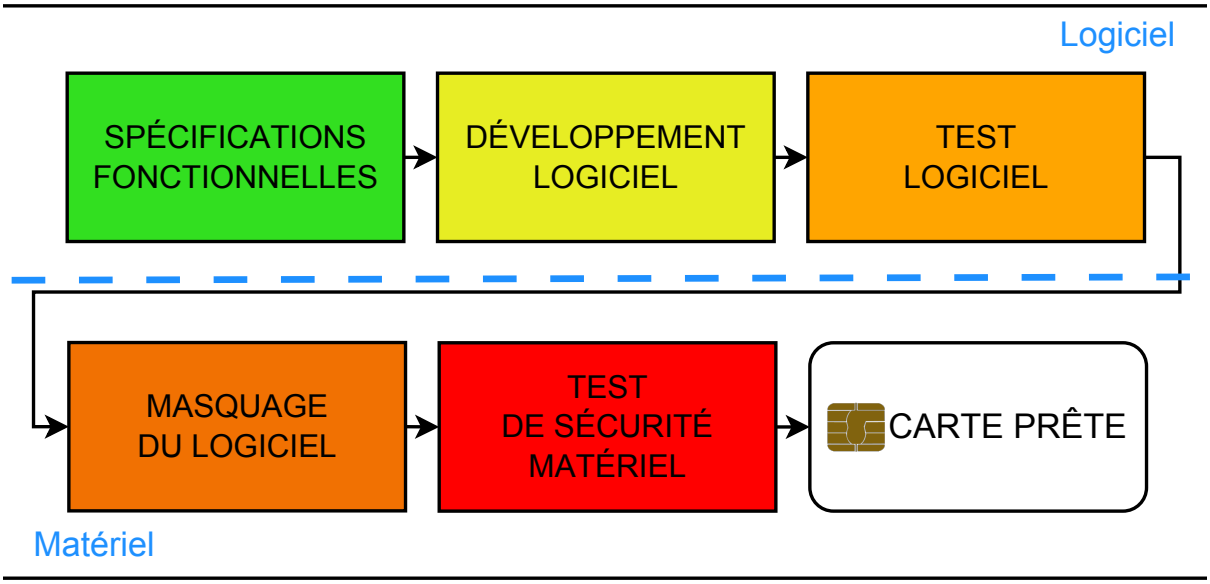


FIGURE 1.5 – Description simplifiée des différentes étapes dans le processus de développement du logiciel d'une carte à puce

On comprend, en regardant l'ordre des différentes étapes de ce processus, pourquoi il est intéressant d'intégrer la vérification de la sécurité tôt au cours de celui-ci afin d'éviter une nouvelle itération de l'ensemble du processus en cas d'un échec des tests sécuritaires physiques. En effet, si les tests de sécurité physique échouent, la carte ne peut pas être produite. La phase de test logiciel ainsi que le déploiement sur composant physique doivent à nouveau être effectués. Cette phase inclut une mise à jour du logiciel pour inclure des contre-mesures contre les attaques découvertes pendant la phase de test physique. Le cycle de développement doit donc reprendre à la phase de développement logiciel. Cette phase terminée, le composant doit être une nouvelle fois évalué et certifié contre les attaques physiques. L'ensemble de ces opérations est extrêmement coûteux en temps et en ressources.

On peut voir dans les figures 1.6 et 1.7 l'avantage que représenterait une solution logicielle permettant d'introduire, pendant la phase de test logiciel, des tests de sécurité simulant les tests matériels effectués plus tard dans le processus. Déplacer une partie de la phase de test matériel en les simulant de manière logicielle permettrait une avance de phase dans la découverte de vulnérabilités. Dans cette thèse, nous cherchons en simulant les attaques physiques au niveau logiciel à obtenir un tel avantage. [Dutertre *et al.* 2009] mentionne aussi le coût élevé de la caractérisation matérielle contre les fautes physiques. Les auteurs proposent notamment une technique de caractérisation basée sur la simulation qui permettrait de réduire l'effort à fournir à cette étape. Basée sur une simulation bas niveau (sur un FPGA) des conséquences d'une attaque modifiant la synchronisation temporelle du circuit, ils arrivent à provoquer des fautes sur des implémentations cryptographiques matérielles. Dans cette thèse, nous suivons le même principe mais souhaitons simuler de manière purement logicielle l'ensemble des attaques par fautes matérielles modifiant la valeur de variable ou le code interprété.

Ce processus présenté ici suit un processus de développement logiciel standard illustré de manière simplifié dans les schémas précédents. Le processus de sécurisation du code intervient

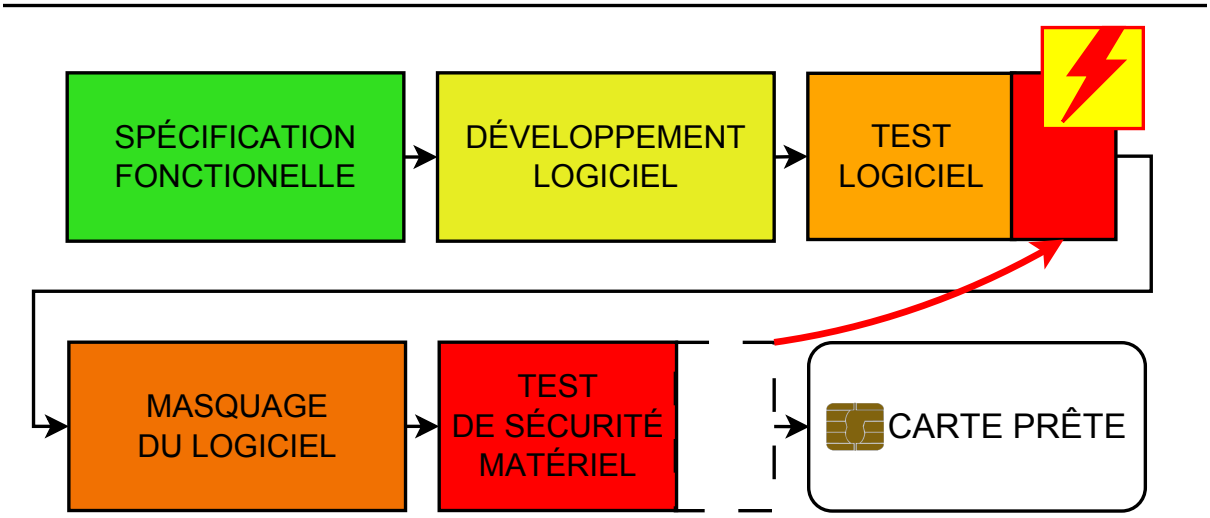


FIGURE 1.6 – Une simulation logicielle des tests de sécurité matérielle entraine une économie de tests physiques plus longs et plus couteux

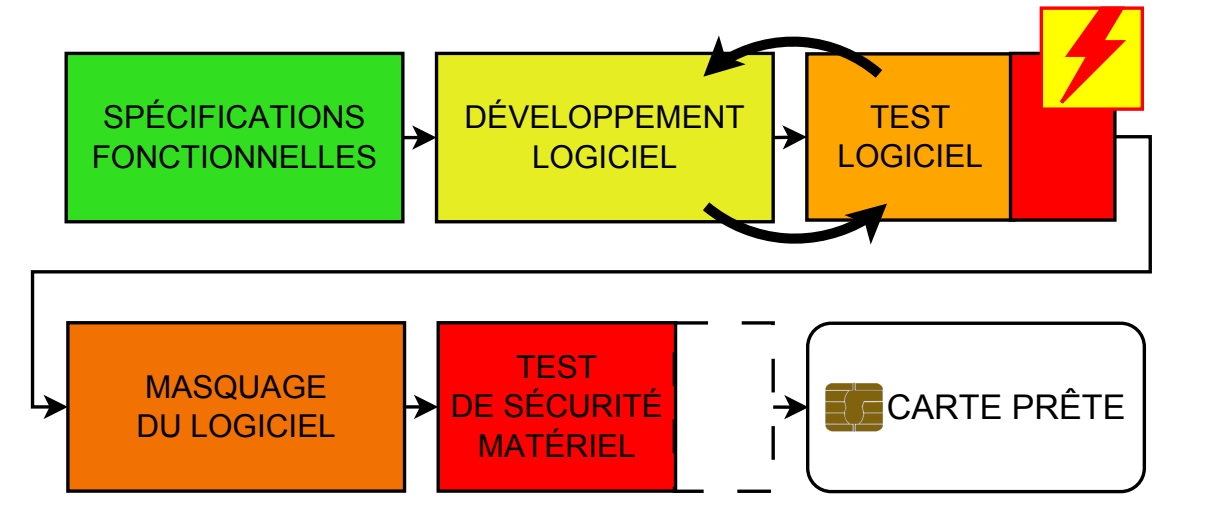


FIGURE 1.7 – Itération préliminaire entre la phase de développement et la phase de test logiciel simplifiant la phase de test matériel

souvent après ou en parallèle des tests logiciels. Ce processus prend la forme d'une revue de code faisant intervenir un spécialiste de la sécurité et le développeur applicatif chargé du développement fonctionnel. Nous prenons en compte ce double point de vue et cette séparation des compétences dans notre approche.

Avant de nous intéresser au fonctionnement interne d'un microcontrôleur qui est à la base de tout système embarqué, nous allons parler des différents types de composants existants, leurs différences. Ces différences ont un impact sur la chaine de compilation et le processus de développement. Nous verrons que certaines de ces différences conditionnent aussi le processus de sécurisation logiciel.

Familles et types de cartes

Familles de carte à puce Il existe différentes familles de cartes à puce.

- Les cartes à mémoire simple servent de simple support de stockage ;
- Les cartes à mémoire avec logique câblée possèdent un support de stockage et sont capables d'effectuer des opérations de logique simple ;
- Les cartes à microprocesseur sont des cartes possédant une mémoire programmable ainsi que des fonctionnalités permettant des opérations logiques avancées.

On s'intéressera aux cartes à microprocesseur dites aussi cartes "intelligentes". Cette "intelligence" vient du fait que contrairement aux cartes à mémoire les cartes à microcontrôleur peuvent être programmées pour effectuer des décisions logiques complexes.

Types de cartes On distingue aussi deux grands types de cartes qui se différencient par la technique utilisée pour communiquer avec le lecteur :

- La carte sans contact, basée sur une technologie RFID, ces cartes permettent de communiquer sans contact physique avec le lecteur. L'alimentation en courant provient néanmoins du terminal ;
- La carte contact nécessite un contact physique direct avec le lecteur au niveau du micromodule afin que l'alimentation et les commandes soient transmises.

Parmi les cartes contact, on note les cartes avec *bandes magnétiques* ou les cartes à puce contenant un *microcontrôleur*. Les cartes à bande magnétique, très déployées, en particulier aux États Unis, sont simples d'utilisation mais d'un niveau de sécurité plus faible par rapport aux cartes à microcontrôleur. Les cartes à puce sans contact évoluent dans un environnement moins sécurisé que les cartes contact et demandent des considérations de sécurité différentes. Dans cette thèse, nous nous intéresserons essentiellement aux cartes contact à microcontrôleur.

Les caractéristiques techniques des différentes cartes à puce sont regroupées en normes et spécifications. Dans la section suivante, nous présenterons ces différentes normes et spécifications existantes qui déterminent les caractéristiques des cartes à puce. Ces normes et spécifications incluent des aspects de sécurité intervenant à différents niveaux (protocolaire, cryptographique, fonctionnel ...) ainsi qu'à des moments différents du processus de création d'une carte (personnalisation, utilisation).

Normes et spécifications

Normes Deux normes internationales régissent les caractéristiques des cartes à puce :

ISO/IEC 14443 pour les cartes sans contact ;

ISO/IEC 7816 pour les cartes contact.

Ces normes décrivent aussi bien les caractéristiques physiques que la syntaxe des commandes à envoyer à la carte ou les schémas cryptographiques à employer. Afin de pouvoir être compatibles avec les différents terminaux existants, les cartes doivent respecter ces normes. Pour plus d'informations sur celles-ci, le lecteur peut consulter les références [Wikipedia 2012c] et [Wikipedia 2012d].

Spécifications Au dessus de ces normes, des spécifications, établies par EMVCo pour la migration et l'interopérabilité des terminaux et cartes à puce, formalisent le fonctionnement applicatif des programmes embarqués dans les microcontrôleurs. La spécification EMV 4.1 est composé de 4 livres définissant notamment les règles à respecter au niveau du protocole de communication afin d'adhérer à la norme EMV. Les terminaux et cartes respectant la norme EMV sont compatibles entre eux. La section 1.2.1 apportera plus de précisions sur la communication entre carte et terminal.

D'un point de vue de la sécurité, [Bond 2006] montre que ces protocoles sont vulnérables à des attaques logiques. [Murdoch *et al.* 2010] montre qu'il est possible d'attaquer des protocoles aujourd'hui mis en place dans l'univers bancaire en utilisant des attaques "man in the middle". Ces attaques connues dans le domaine des réseaux repose sur le fait qu'un attaquant malveillant s'insère entre deux interlocuteurs légitimes et parvient à espionner et / ou modifier l'échange entre ces interlocuteurs. Ces attaques montrent qu'il est important de s'intéresser à toute la chaine de sécurisation et pas seulement à un point en particulier. Elles valident aussi la transposition d'attaques de domaines différents au monde de l'embarqué. Dans cette thèse, nous nous intéressons aux attaques sur le code source embarqué dans les cartes à puces. Cependant, nous ne nous focalisons pas sur les particularités protocolaires ou cryptographiques de ces codes, nous considérons tout code comme une implémentation d'un objectif fonctionnel qui peut être mis en défaut.

Communication avec une carte

Avant d'aller plus loin, nous introduisons quelques termes utilisés dans la communication entre une carte et un lecteur. Tout d'abord le terminal permet de communiquer avec la carte et de lui envoyer des instructions sous forme de commandes. Chacune de ces commandes remplit un objectif particulier contribuant à un objectif final. L'ensemble de ces commandes constitue une transaction débutant au moment où la carte est alimentée et s'achevant lorsqu'elle ne l'est plus. Une transaction permettra à l'utilisateur de la carte d'effectuer une à plusieurs tâches nécessaires pour remplir l'objectif premier de la carte, par exemple s'authentifier auprès d'un organisme pour prouver son identité et autoriser une demande.

Afin de communiquer avec une carte, on doit disposer d'un lecteur. Ce lecteur sert d'interface entre la carte et le terminal ; il va transmettre des commandes à la carte en respectant un certain protocole. Le dialogue se fait par "trames" de communication appelées APDU. Les standards ISO/IEC 7816 ou ISO/IEC 14443 et la norme EMV définissent les commandes à envoyer pour établir la communication entre le terminal et la carte. Deux protocoles existent :

T=0 est le plus utilisé et utilise une transmission de donnée par caractère ;

T=1 est moins utilisé, il utilise une transmission de donnée par blocs structurés.

Ces protocoles sont décrits en détail à l'intérieur de la norme ISO 7816-3.

L'amorce d'une connexion entre la carte et le terminal se fait par une réponse à un signal électrique émis sur le contact RST du micromodule. La carte renvoie une réponse normalisée ainsi que les informations nécessaires au terminal pour établir la communication (par exemple : la vitesse d'échange).

Une session de communication correspond à un échange de trames entre le terminal et la carte. Le terminal utilisera plusieurs commandes afin de déclencher certaines actions. La liste ci-dessous résume les principales étapes de la transaction :

1. Application selection
2. Initiate application processing
3. Read application data
4. Processing restrictions
5. Offline data authentication
6. Cardholder verification
7. Terminal risk management
8. Terminal action analysis
9. First card action analysis
10. Online transaction authorisation
11. Second card action analysis
12. Issuer script processing

Ces étapes se traduisent en un certain nombre de commandes qui peuvent être envoyées à la carte :

Power On le terminal carte fournit du courant à la carte et commence le processus d'initialisation. La carte répond par un Answer To Reset ;

Select Application le terminal sélectionne une application ;

Get Processing Options le terminal demande des informations à la carte sur son contenu ainsi que des données fonctionnelles pour le reste de la transaction ;

Read Record le terminal va lire aux endroits appropriés dans le système de fichier des informations supplémentaires sur l'application sélectionnée ;

Get Challenge le terminal demande à la carte un nombre aléatoire à la carte pour diversifier ou chiffrer une opération future ;

Internal authenticate authentifie l'application pour que le terminal puisse accéder à des données sensibles de la carte ;

Verify Pin le porteur est authentifié auprès de la carte ;

Generate Application Cryptogram un cryptogramme d'autorisation est généré et la transaction finalisée.

Pour plus de précision sur le fonctionnement et les détails de chaque étape, le lecteur pourra se référer à la norme EMV. Une explication simplifiée peut aussi être trouvée dans le document [Bouzefrane 2008]. Pour résumer, on peut dire que ces commandes décomposent un échange permettant une authentification de la carte et du porteur dans une transaction décrite dans la section 1.1.4.

Depuis son apparition, le protocole EMV a été la cible de nombreuses attaques. Différents points de vulnérabilité du protocole peuvent être trouvés à la référence [Wikipedia 2012a]. Ce protocole sert aussi de point d'entrée à de nombreuses attaques de type logiciel. [Bond & Anderson 2001] et [Buetler 2008] montrent deux types d'attaques pouvant arriver au niveau protocolaire à compromettre la sécurité en utilisant des approches différentes. En envoyant des séquences de commandes dans le désordre ou en espionnant les trames d'échange, un attaquant peut mettre en évidence des vulnérabilités au niveau logiciel. Une autre approche est adoptée par [Lancia 2011] où une technique de fuzzing est utilisée contre les paramètres des commandes envoyées afin de faire ressortir des vulnérabilités dans le code implémentant le protocole.

Ces approches montrent qu'au niveau protocolaire, l'implémentation est source de vulnérabilité et qu'il est nécessaire de disposer de techniques et d'outils afin de vérifier cette implémentation logicielle contre les différents types d'attaques existants.

Une des difficultés à mettre en place une solution et des outils permettant une vérification générique de la sécurité provient de la diversité des composants utilisés et donc des chaines de compilation utilisées en embarqué.

Chaine de compilation et généricité

Les fondeurs peuvent fournir leur propre chaine de compilation et environnement de développement ou s'appuyer sur des environnements existants tels que KEIL. Ils peuvent aussi choisir d'implémenter des instructions spéciales pour leur composant, d'adopter un agencement mémoire particulier ou de supporter des optimisations spécifiques. A cause de ces différences, il est difficile pour le développeur et la personne chargée de la sécurité de trouver une méthode générique applicable sur l'ensemble des composants pour sécuriser leur application. Cependant l'ensemble de ces fondeurs supportent généralement l'ensemble du standard C ANSI. Ainsi, il est plus facile d'incorporer des sécurités au niveau du code source avant la compilation par l'environnent spécifique au fondeur. Les sécurités logicielles à incorporer demandent de la part des spécialistes en sécurité une première phase d'évaluation du composant face aux attaques physiques. La résistance du composant à cette phase d'évaluation définira le besoin de renforcer la sécurité au niveau logiciel. Souvent, pour des raisons de coût, des composants avec peu de défenses matérielles sont choisis et une grande partie de la sécurité est déléguée au niveau logiciel.

1.2.2 Fonctionnement interne

Après avoir parlé des étapes de fabrication d'une carte, nous allons nous intéresser à son fonctionnement interne afin de mieux comprendre comment une attaque physique peut affecter le fonctionnement logiciel.

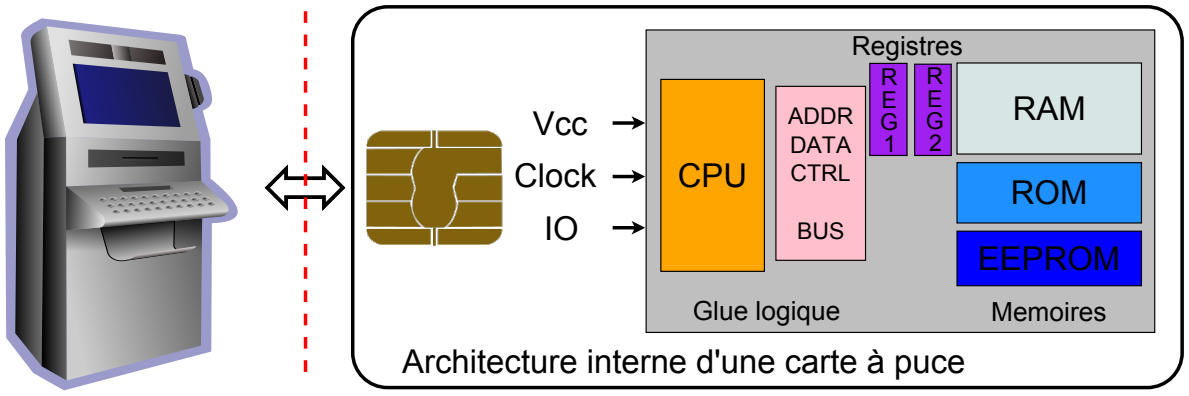


FIGURE 1.8 – Schéma simplifié d'une carte

Composants essentiels d'une carte à puce

Une carte à puce est physiquement composée de trois éléments :
- La carte plastique ;
- Le micromodule ;
- La puce.

La carte plastique est le socle physique de base de la carte. Sur celle-ci se trouve la puce en elle même qui contient les données et le programme à exécuter. La puce se trouve sous le micromodule seule partie visible depuis l'extérieur et dont les contacts servent d'interface avec le lecteur.

La figure 1.8 montre un schéma simplifié de la carte en faisant apparaitre les différents composants entrant en jeu : les composants mémoires, les bus et le microprocesseur.

Caractéristiques techniques d'une carte à puce

Dans cette section, nous allons parler des caractéristiques d'une carte à puce. Nous allons passer en revue les différents composants physiques et expliquer leur fonctionnement. Dans un deuxième temps, nous prendrons l'hypothèse d'une attaque physique et observerons les conséquences de celle-ci sur le fonctionnement de la carte.

Architecture d'une carte à microprocesseur Une carte ressemble schématiquement à la figure 1.9.

Dans la figure 1.9 on peut voir les différents composants matériels formant une carte.

On distingue les différents blocs matériels internes :
- Interfaces externes ;
- Alimentation et horloge ;
- Unité centrale et microprocesseur ;
- Les différentes mémoires externes ROM/ RAM/ EEPROM ;
- Le bus interne de communication.

Chacun de ces éléments peut servir de cible à une attaque physique. Ces vecteurs d'attaque sont des moyens pour l'attaquant d'influer sur le système : ils font souvent partie des prérequis

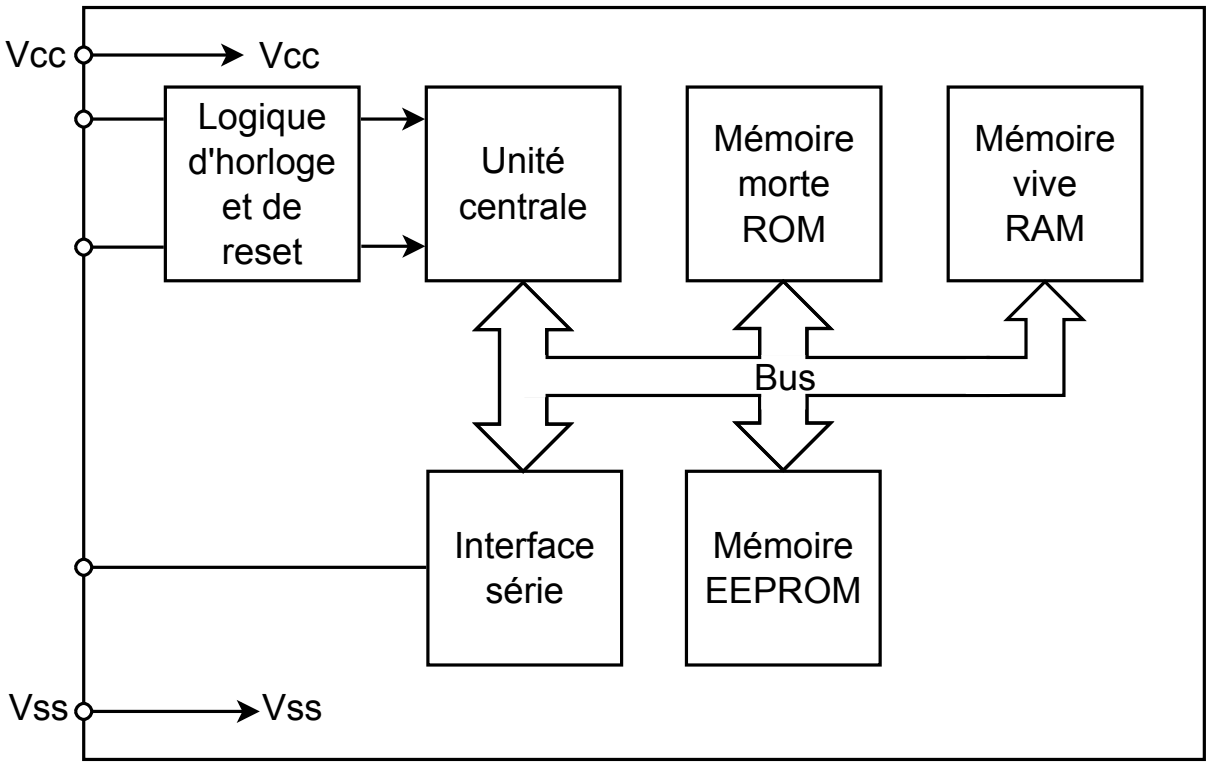


FIGURE 1.9 – Schéma d'une carte [Consultants 2009]

nécessaires au bon fonctionnement de l'architecture du microcontrôleur ou d'un des composants clés responsables d'une tâche spécifique nécessaire à l'ensemble de la carte.

Interfaces externes Les interfaces externes permettent à la carte de communiquer avec le terminal. Elles alimentent aussi celle-ci et fournissent les signaux électriques responsables des changements électriques au sein de la puce. Ces changements électriques et leur synchronisation sont à la base de tout échange d'information et de logique du système.

On retrouve sur le micromodule les plaques de contact donnant accès aux fonctionnalités mentionnées ci-dessous.

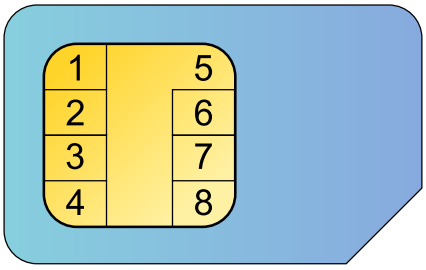


FIGURE 1.10 – Illustration du micromodule

(1) Vcc le signal de l'alimentation en courant externe

(2) RST le signal de redémarrage de la carte

(3) CLK le signal d'horloge externe (beaucoup de composants disposent maintenant d'une horloge interne permettant de s'affranchir d'un signal externe pouvant introduire des problèmes de sécurité

(4) et (8) RFU configurable suivant l'utilisation

(5) GND la masse

(6) Vpp la tension de programmation

(7) I/O le signal d'entrée sortie qui sert à encoder les messages en provenance ou en direction du terminal

L'alimentation est facilement accessible par un attaquant qui peut brancher une alimentation dont il contrôle le voltage. En alimentant la carte avec un voltage excédant le seuil admis pour le fonctionnement de la carte, un attaquant peut espérer perturber le fonctionnement de celle-ci à son avantage. Il existe aussi un risque permanent que l'alimentation de la carte soit coupée à tout moment effaçant ainsi toutes les données en mémoire volatile et interrompant les opérations en cours ou des écritures de données qui auraient dû avoir lieu. Le séquencement des commandes et la manière dont celles-ci sont implémentées est donc très important si on veut garder une cohérence des données mais aussi une cohérence sécuritaire du système. Des mécanismes sont donc responsables du respect de l'intégrité des données en cas de coupure de courant. Il est cependant nécessaire de vérifier l'intégrité sécuritaire du système. Des capteurs de sous-tension et de surtension sont maintenant mis en place pour se protéger contre de telles attaques.

L'horloge constitue aussi un vecteur d'attaque de choix pour un attaquant. Une lecture ou écriture est généralement synchronisée sur le front montant ou descendant d'une horloge. A ce moment précis, la valeur présente sur un ou plusieurs bus sont lus afin d'être transmise. Ainsi en ralentissant ou accélérant l'horloge, un attaquant peut espérer fausser la valeur qui sera transmise. La plupart des composants récents embarquent une horloge interne pour se protéger contre de telles attaques.

Les données de la carte sont transmises et reçues par l'entrée I/O. Un attaquant externe, du point de vue de la carte, est capable d'observer, de contrôler ou de modifier les données transitant sur cette entrée. On estime donc qu'il est capable de choisir les commandes envoyées à la carte. Cette position lui permet de monter des attaques dites "logiques" en exploitant l'interface d'entrée I/O. Ces attaques logiques sont à la base des attaques contre le protocole mentionnées dans la section 1.2.1. Certaines des attaques mentionnées dans [Criteria 2009] sont les attaques logiques qui exploitent des failles existantes dans l'implémentation faite des spécifications. Si ces attaques ne nécessitent pas d'induire d'erreurs au niveau matériel, une perturbation physique peut être considérée comme un facteur aggravant qui, combinée aux attaques logiques, peuvent mettre à jour de nouvelles vulnérabilités.

On notera aussi les attaques par découverte de fonctions de test oubliées par le développeur. L'attaquant enverra toutes les suites possibles de commandes à la carte en espérant que celle-ci réponde lors d'une commande non spécifiée. Il aura ainsi découvert une nouvelle entrée logicielle pouvant potentiellement lui servir de vecteur d'attaque. On remarque l'intérêt des méthodes formelles dans ce contexte permettant de prouver, notamment dans les protocoles de communication, que ceux-ci sont exempts de failles sécuritaires.

Différents types de mémoire Différents types de mémoire cœxistent sur un composant embarqué. Le mode de rétention de l'information, caractéristique principale de chacune d'elles, conditionne la mécanique interne (logique et matérielle) de ce type de mémoire. On distingue les mémoires non volatiles (EEPROM), des mémoires volatiles (RAM). Leur utilisation permet la mémorisation d'informations pour une utilisation future. L'intérêt pour l'attaquant ainsi que le procédé d'attaque sont différents suivant le type de mémoire impliquée dans l'opération matérielle attaquée. Les effets des attaques auront une durée plus ou moins longue si la variable affectée est écrite en EEPROM ou en RAM. On peut aussi noter que certaines informations sont écrites de manière définitive en ROM dans la carte au moment de sa fabrication.

De manière générale, les écritures fonctionnelles en EEPROM qui interviennent au cours de la transaction sont critiques pour le bon fonctionnement sécuritaire de la carte. Ainsi, les interfaces de lecture et d'écriture en mémoire EEPROM sont des positions d'attaque privilégiées. Dans le cas d'une mise à jour d'un compteur de sécurité, par exemple le nombre d'essais du code PIN, réussir à empêcher la mise à jour de ce compteur en empêchant son écriture représente un gain pour l'attaquant.

Les mémoires volatiles contiennent des données dont la durée de vie est conditionnée par l'alimentation en courant du microcontrôleur. En cas de coupure, les informations contenues dans cette mémoire sont perdues. Certaines variables globales peuvent cependant avoir une durée de vie dépassant la durée d'un échange APDU. Celles-ci peuvent être utilisées pour conserver une information temporaire, nécessaire au long d'une transaction, mais ne demandant pas une écriture en mémoire EEPROM en vue d'une utilisation future.

Les interfaces matérielles et logicielles manipulant la mémoire dans un contexte de sécurité sont donc très sensibles et doivent être sécurisées contre les attaques par fautes. Dans un avenir proche, la généralisation de la mémoire FLASH dans les composants embarqués de type carte bancaire apportera de nouveaux défis de sécurité.

Les moyens à la disposition des attaquants pour perturber le matériel sont donc nombreux. Ils reposent essentiellement sur un apport d'énergie supplémentaire et localisé à une partie du circuit. Un apport de lumière, une perturbation du courant électromagnétique, une hausse de la température peuvent être utilisés pour modifier le comportement du logiciel interprété sur le matériel attaqué. Nous traiterons en détail, dans la section 3.2.2 du chapitre 3, des capacités d'action des attaquants leur permettant de perturber les composants matériels notamment les mémoires.

Architecture Harvard vs Von Neumann Dans ce paragraphe, nous discutons de l'architecture mémoire utilisée pour transmettre des informations stockées en mémoire volatile ou non volatile jusqu'au microprocesseur où elles doivent être interprétées. Par défaut, un microcontrôleur 8051 adopte une architecture Harvard, c'est-à-dire que le code et les données proviennent de deux espaces mémoires différents (ROM et RAMEEPROM). L'architecture peut cependant être modifiée pour adhérer à l'architecture Von Neumann où le code et les données proviendront du même espace mémoire. Des données seront ainsi interprétées comme du code. Un tel changement permet de charger du code exécutable dans une zone mémoire programmable (EEPROM) et non dans une zone en lecture seule comme la ROM. Cette fonctionnalité peut être utilisée pour charger du code exécutable a posteriori de la création de la puce. En effet, une fois le code embarqué en ROM celui-ci ne peut plus être modifié.

Un fonctionnement qui respecte l'architecture Von Neumann peut poser des problèmes de

sécurité au moment du chargement du code exécutable. Le fait que du code exécutable peut être interprété pendant la phase d'utilisation de la carte permettrait à un attaquant, s'il arrivait à charger du code malicieux, de créer une faille de sécurité. De plus, avec une double attaque, il pourrait à la fois exploiter une faille dans le code original et en même temps à contourner le code correctif rajouté.

Types de plates-formes embarquées On distingue deux types de plates-formes sur lesquelles sont basées les cartes à puce :

Les plates-formes natives Développées en C, elles coutent relativement peu cher et leur principal avantage est leur rapidité. Elles sont beaucoup utilisées dans le système bancaire et ont une architecture monolithique qui ne permet pas l'ajout de nouvelles applications une fois déployées.

Les plates-formes ouvertes Souvent développées en Java leur principal avantage est une grande interopérabilité / portabilité sur la base d'une machine virtuelle et d'applets Javacard. Leur désavantage réside dans le fait qu'elles sont plus onéreuses ainsi que plus lentes.

Par la suite, nous traiterons spécifiquement des plates-formes natives qui est le sujet principal de cette thèse. Cependant, certains résultats théoriques obtenus peuvent s'appliquer sur tout système embarqué à base de microcontrôleur.

Systèmes d'exploitation Le système d'exploitation est une partie importante d'un système embarqué. Ce système est construit selon une architecture logicielle donnant accès aux capacités matérielles du composant. Cette architecture structurée fournit des services génériques nécessaires au bon fonctionnement logiciel de la carte. Parmi celles-ci, on peut citer :

Le système de fichier responsable de l'agencement des informations en mémoire EEPROM, ce système de fichier est construit autour de dossiers contenant les informations d'une même application et de fichiers, des conteneurs unitaires pour ces informations ;

Les briques cryptographiques qui fournissent des services de chiffrement, déchiffrement, signature mentionnés dans la section 1.1.3 ;

Les mécanismes de journalisation responsables de l'intégrité des processus d'écriture en EEPROM en utilisant une écriture différée dans une zone mémoire tampon ;

Les mécanismes d'erreurs responsables de la gestion des erreurs logicielles ou matérielles qui peuvent apparaître dans un contexte fonctionnel normal et fournissant au terminal les informations sur le contexte et éventuellement l'origine de l'erreur.

Le choix de l'architecture logicielle et l'intégration de la sécurité au moment de la conception du système d'exploitation sont particulièrement importants afin de garantir une sécurité contre les attaques physiques à travers l'ensemble du système.

Microcontrôleurs

La base électronique d'une carte à puce est le microcontrôleur. Dans cette section, nous détaillerons son architecture interne afin de comprendre comment interagissent les différentes parties matérielles et comment une attaque peut réussir à les perturber.

Architecture détaillée Un microcontrôleur est composé d'un microprocesseur et de mémoires. Le microprocesseur peut être décomposé en une unité de calcul (UC), une unité arithmétique et logique (UAL) ainsi que des registres dont l'accumulateur ACC et les registres d'états (C,AC,S,O,Z,P). Des registres spéciaux SFR sont dédiés au contrôle des périphériques raccordés au microcontrôleur.

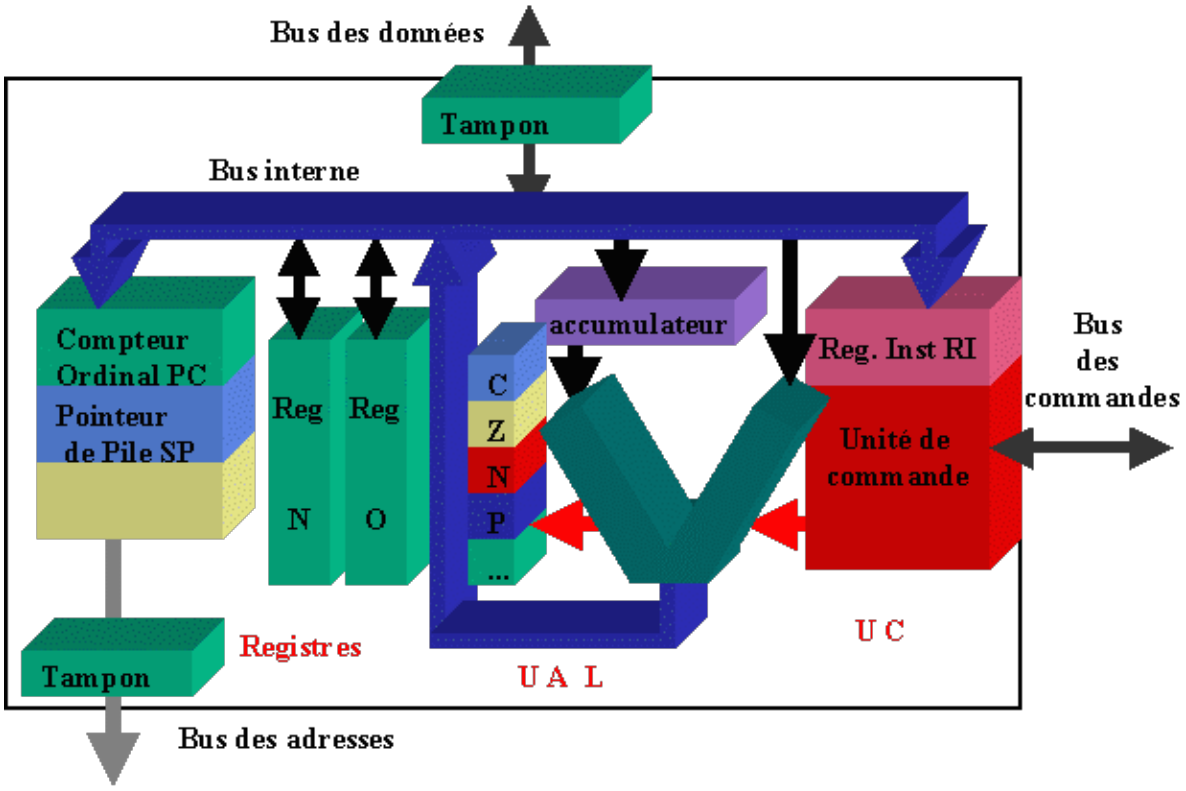


FIGURE 1.11 – Architecture interne du microprocesseur [Litwak 1999]

La figure 1.11 illustre les composants internes du microprocesseur. On distingue les différents éléments responsables de l'interprétation du code.

L'UC, responsable de l'exécution du code programmé en mémoire ROM, suit un fonctionnement en 4 étapes :

Fetch l'instruction à exécuter est chargée depuis un registre spécifique, le registre d'instruction RI ;

Decode l'instruction est décodée par rapport au jeu d'instructions du composant ;

Exec cette instruction est ensuite exécutée entrainant les modifications correspondantes dans le reste du système ;

Prep Next finalement, la prochaine instruction est préparée ce qui équivaut à déplacer le curseur de lecture du code du programme en ROM.

Attaquer une de ces étapes peut avoir un effet sur le code qui sera exécuté. L'UC fonctionne aussi avec l'aide de l'UAL responsable des opérations arithmétiques (addition, soustraction,

multiplication ou division) et logiques (et, ou, négation logiques) dont a besoin le programme pour effectuer des calculs ou des combinaisons logiques. Attaquer directement l'UAL pourra avoir des conséquences sur le résultat d'un calcul ou une décision logique. De même, les conséquences de l'attaque d'un registre dépendra de son utilisation. En effet, modifier un registre contenant la valeur intermédiaire d'un calcul changera le résultat de ce calcul. Alternativement, modifier un registre d'état influencera une instruction suivante comme, par exemple, un saut conditionnel qui se base sur cet état pour prendre une décision.

Mémoires Les mémoires correspondent à des cellules capables de retenir un état électrique. Ces cellules sont regroupées en cases mémoires dont la taille dépend de l'architecture et qui peuvent contenir une information.

La ROM contient le code du programme ainsi que les données permanentes de celui-ci. L'index permettant de parcourir cette mémoire est le Program Counter (PC). La taille standard de la ROM sur un 8051 est de 4K ;

La RAM contient les données du programme à l'exécution, la pile accessible par le Stack Pointeur (SP) et 32 registres d'un octet chacun. Les 128 octets de la RAM peuvent être adressés en utilisant le Data Pointeur (DPTR) d'une longueur de 16 bits ;

L'EEPROM contient les données permanentes modifiables du système. On y accède généralement à l'aide d'un système de fichiers.

Attaquer ces différentes mémoires peut avoir un effet différent suivant leur type et l'utilisation qui en est faite par le programme. On peut noter que la RAM reste alimentée dans un scénario normal tout au long de la transaction. Ainsi, les informations résidant en RAM, si elles sont perturbées, auront un effet qui peut perdurer jusqu'à la fin de la transaction. Modifier une donnée écrite en EEPROM aura un effet qui ne se limite pas à une simple transaction mais peut influencer les transactions futures. Des informations sensibles concernant l'état de la carte sont généralement gardées en EEPROM. Attaquer ces informations au moment de leur écriture en EEPROM peut être avantageux pour un attaquant. C'est pourquoi les interfaces d'écriture en EEPROM (ainsi que les interfaces de lecture) doivent être soigneusement sécurisées contre les attaques physiques. Les contre-mesures visant à défendre l'intégrité des informations stockées en EEPROM sont souvent basées sur des mécanismes de hachage tels que des checksums. Ces checksums sont calculés et sauvegardés en EEPROM en même temps que les informations sensibles s'y trouvant et vérifiés après chaque écriture.

Bus Les bus sont des points d'attaque privilégiés par les attaquants qui utilisent les fautes physiques. En effet, les bus se trouvent à la périphérie des zones mémoires facilement reconnaissables sur les composants. Ces zones de transfert d'information sont donc ciblées par les attaques physiques car elles sont faciles à localiser sur la puce et présentent un gain élevé pour l'attaquant.

La figure 1.12 montre les différents composants mémoire d'un microcontrôleur. On distingue nettement les différentes zones mémoires ainsi que le microprocesseur. Les bus se trouvent à la limite des zones mémoires.

On distingue 4 types de bus :

Le bus de données de 8 à 128 bits, ce bus bidirectionnel permet la communication entre l'UC et la mémoire en passant par le RI, l'ACC ou les registres ;

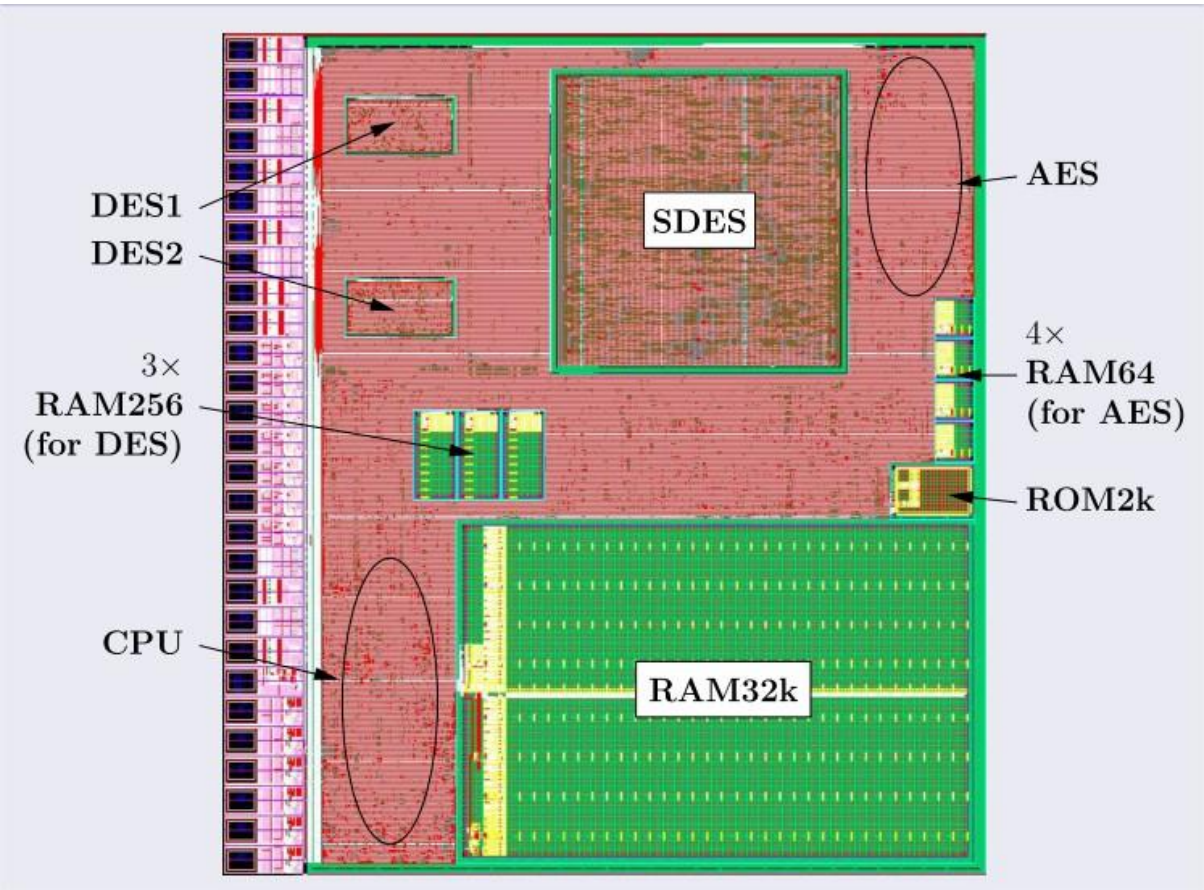


FIGURE 1.12 – Vue d'une puce et des différents composants mémoire [Guilley 2007]

Le bus d'adresse sur 16 bits, ce bus unidirectionnel permet à l'UC de spécifier des adresses aux registres ;

Le bus de contrôle ou de commande bidirectionnel, il met en relation l'UC et les mémoires au travers de commandes de lecture et d'écriture ;

Le bus interne du microprocesseur met en relation l'UC, l'UAL, l'ACC, les registres, les mémoires tampons du microprocesseur et le bus de données.

Attaquer ces bus aura pour effet de perturber les valeurs de variables écrites dans les différentes zones mémoires. L'effet peut aussi fausser un calcul ou faire en sorte qu'un code différent soit interprété par le composant. Ainsi attaquer le bus de données au moment du Fetch aura une conséquence sur l'instruction décodée et exécutée. Attaquer ce même bus à un autre moment pourra avoir un effet différent ou ne pas avoir d'effet. Certaines attaques peuvent aussi être équivalentes d'un point de vue matériel. Ainsi perturber un bus lors du Fetch aura pour conséquence qu'une valeur faussée sera chargée dans le RI. Cependant perturber directement le RI pourra avoir le même effet.

La compréhension des mécanismes de fonctionnement interne d'un microcontrôleur sont nécessaires pour évaluer au niveau logiciel les conséquences d'une attaque.

1.3 Besoins en sécurité

Après avoir expliqué les mécanismes de fonctionnement interne du matériel sur lequel s'exécute le code d'une carte à puce, nous proposons d'aborder les besoins en sécurité. Nous avons montré qu'attaquer physiquement un composant peut perturber l'exécution du programme, voyons maintenant comment des attaquants ont su tirer profit de ces attaques pour créer et exploiter des vulnérabilités logicielles et pourquoi il est nécessaire de s'en protéger.

1.3.1 Historique de la sécurité des cartes

Les attaques sur carte à puce ont fortement évolué depuis leur apparition. La puissance de calcul disponible aujourd'hui force l'utilisation de clés de chiffrement de tailles toujours plus grandes. De nouveaux algorithmes cryptographiques (RSA, AES, courbes elliptiques) ont été déployés sur celles-ci. Des attaques de plus en plus sophistiquées ont fait leur apparition [van Woudenberg *et al.* 2011] tandis que les constructeurs et les développeurs cherchent des moyens de plus en plus ingénieux afin de sécuriser leur code au niveau matériel [Guilley *et al.* 2008] comme au niveau logiciel [Guilley *et al.* 2010].

La carte à puce, invention de Roland Moreno et Michel Ugon en 1974 a cependant su s'imposer comme l'élément clé de la sécurité de nombreux système d'authentification. En tant que tel, elle a subi de nombreuses attaques depuis sa création jusqu'à aujourd'hui.

1996 A la conférence Usenix, les attaques sur les cartes à puce sont abordées afin de montrer que leur sécurité peut être compromise et donc que les systèmes basés sur celles-ci peuvent être mis en danger [Anderson & Kuhn 1996] ;

1996 Les premières attaques par canaux cachés sont mises en œuvre sur des algorithmes cryptographiques en exploitant des informations acquises en mesurant les temps d'exécution [Kocher 1996] ;

1997 Les premières attaques concrètes sont décrites ;

2000 L'affaire Serge Humpich en France et la célèbre Yes Card permet de faire croire à un terminal que la carte renvoie une réponse correcte à une demande d'authentification. Une explication de l'attaque peut être trouvée à la référence [Cortier 2005] ;

2000 En parallèle, à la conférence Usenix, de nombreuses attaques sont envisagées où la carte à puce est utilisée comme vecteur de vulnérabilité [Gobio *et al.* 2000] ;

2003 Les premières attaques optiques sont décrites ainsi que leur application dans le but de compromettre la sécurité des cartes à puce [Skorobogatov & Anderson 2002] ;

2010 Christopher Tarnovsky à la conférence Black Hat montre un exemple d'attaque invasive ainsi que les possibilités de "mettre à nu" une carte à puce et de la "retro ingénierer" [Tarnovsky 2010] ;

2010 En parallèle, Ross Anderson à Cambrige décrit diverses attaques contre le protocole qui régit les échanges entre les terminaux et les cartes [Murdoch *et al.* 2010].

La progression de ces attaques montre que les attaquants sont capables d'évoluer pour trouver de nouveaux moyens de perturber les systèmes à base de carte à puce. Ceux-ci empruntent des chemins de plus en plus indirects et complexes pour provoquer des erreurs et exploiter des vulnérabilités résultantes. C'est pourquoi sécuriser indépendamment les différentes parties d'un programme embarqué ne permet pas d'assurer la sécurité de l'ensemble de celui-ci. Considérer le système dans son intégralité en établissant et en validant une politique de sécurité cohérente sur la totalité du système est nécessaire pour déjouer ces attaques.

1.3.2 Attaques physiques

Dans cette section, nous présentons l'architecture d'un microcontrôleur dans l'hypothèse d'une attaque physique. Les composants matériels de l'architecture d'un microcontrôleur sont ainsi passés en revue. Ce chapitre se termine par le classement de ces attaques en catégories pertinentes vis-à-vis de leur modélisation et plus tard de leur simulation.

Les attaquants ont des compétences différentes. De l'utilisateur final, sans compétence informatique, aux organisations criminelles, disposant d'une équipe d'ingénieurs à l'expertise élevée et de moyens conséquents, comme par exemple des FIBs, en passant par l'étudiant en électronique, disposant des moyens offerts par son université, des attaquants aux compétences et moyens variés vont s'attaquer aux défenses des cartes et chercher à les compromettre.

La figure 1.13 montre un environnement matériel permettant de monter des attaques physiques sur une carte à puce à l'aide d'un laser.

Caractérisation des attaques On peut classifier les attaques suivant le procédé utilisé pour les mettre en œuvre. Ainsi, on distingue des attaques :

Passives qui ne nécessitent pas de modifier physiquement la carte ou la puce ;

Semi invasives qui nécessitent une modification physique non destructive de la carte ou de la puce ;

Invasives qui nécessitent une modification destructive de la carte ou de la puce.

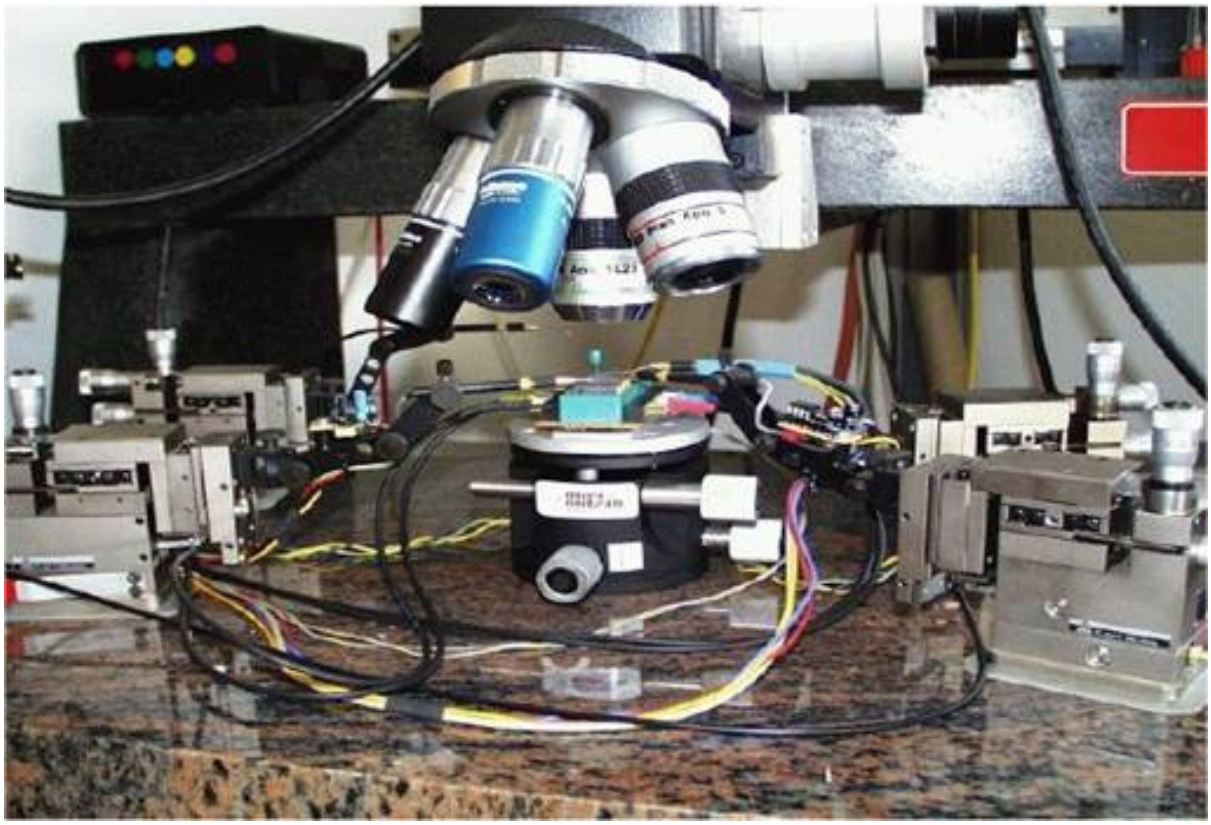


FIGURE 1.13 – Banc d'attaque laser, source : Oberthur Technologies

Dans cette thèse, on s'intéresse aux attaques semi invasives ou invasives de type attaque par pulse laser. L'approche adoptée permet cependant de modéliser toute attaque créant une faute logicielle.

[Bar-El 2003] mentionne des exemples d'attaques classées suivant des catégories similaires : invasives et non invasives. L'auteur donne des détails sur la zone mémoire impliquée dans une attaque. Ces détails montrent que les attaques documentées dans la littérature ciblent l'ensemble des mémoires existantes mentionnées dans le paragraphe 1.2.2.

On peut regrouper différemment les attaques en deux grandes catégories selon que leurs conséquences logicielles affectent une information ou le code la manipulant. On peut ainsi distinguer les attaques qui affectent les valeurs des variables des attaques qui affectent la logique du programme comme les conditions ou les boucles. Dans cette thèse, nous adoptons une telle classification étant donné que cette représentation se prête mieux au travail de modélisation que nous souhaitons mener.

Principe général Les attaques visent à compromettre la carte en permettant la divulgation d'informations secrètes (par exemple, des clés de chiffrement), la remise en cause de l'intégrité de certaines données (par exemple des compteurs d'essais ou même les clés de chiffrement), la remise en cause de l'intégrité du fonctionnement de la carte (par exemple, un déni de service, mais aussi

un changement de comportement) ou le rejeu (une fois certaines informations acquises, il est possible de créer un clone de la carte). Le but de ces attaques est la génération d'argent ou le vol d'identité. Il se résume par une élévation de privilège pour l'attaquant.

Moyens d'attaque Les vecteurs d'attaque sont nombreux : la température, les modifications temporelles, de courant, l'utilisation de radiation ou de lumière. De nombreux facteurs physiques peuvent ainsi être exploités pour créer une faute logicielle. On notera aussi qu'un attaquant, s'il peut contrôler l'environnement dans lequel une transaction est effectuée, peut facilement arracher la carte du lecteur en coupant brutalement son alimentation. Cette coupure aura pour effet d'arrêter l'exécution du code et éventuellement de déclencher certaines sécurités. Un attaquant peut exploiter cette coupure pour réitérer ses tentatives d'attaque. Pour parer à cette éventualité, la journalisation des réponses sécuritaires doit être soigneusement prise en compte lors du développement.

1.3.3 Problématique globale

Face aux besoins de sécurité exprimés dans les sections précédentes, différents moyens de sécurisation doivent être mis en place. Ces moyens de sécurisation doivent être capables de couvrir l'ensemble du système. En effet, sur un système impliquant plusieurs briques logicielles, la sécurité est soumise au problème du maillon faible, c'est-à-dire qu'elle repose sur la résistance de l'élément le moins sécurisé.

L'architecture logicielle d'un système embarqué est généralement composée de différentes couches logiques (bios, systèmes d'exploitation, applications, briques cryptographiques). Ces couches logiques sont assemblées de manière à accomplir les différentes tâches fonctionnelles demandées par les spécifications. Ces tâches fonctionnelles décrites dans les spécifications applicatives sont traduites en un programme qui doit s'exécuter sur une plate-forme matérielle. Or, tout système décomposé de cette manière est soumis à une problématique de visibilité. Certaines de ces couches vont abstraire la complexité des opérations sous-jacentes et proposer d'avoir accès au service par le biais d'interfaces de programmation. On notera aussi que ces différents services sont généralement développés par des personnes différentes, ce qui augmente le risque de voir apparaître des erreurs au niveau des interfaces.

Du point de vue de la sécurité, cette séparation architecturale entre les différents blocs de services entraine une diminution dans la visibilité sécuritaire globale du système. De plus, ces blocs peuvent avoir des considérations en sécurité différentes. Tel bloc ne devra pas laisser fuir d'informations confidentielles tandis qu'un autre bloc devra être sécurisé en terme d'intégrité. Ce manque de visibilité et cette hétérogénéité de la sécurité sur l'ensemble du système rend le travail de la personne en charge de la sécurisation plus difficile.

Il convient par ailleurs de considérer la sécurité de l'ensemble du système et pas seulement certains services réputés sensibles. Par exemple, si un mécanisme de chiffrement existe dans un bloc cryptographique de bas niveau mais qu'une fonction de haut niveau ne l'utilise pas ou l'utilise incorrectement afin de chiffrer des données sensibles, c'est la sécurité de l'ensemble du système qui peut être compromise. De plus, dans le cas d'attaques par faute, il faut aussi s'assurer que le service appelé est correctement exécuté (que les données sont bien chiffrées). Les interfaces de programmation de lecture et d'écriture (boucle de lecture dans un tableau) constituent un autre exemple de point d'attaque. Si celles-ci sont compromises, il serait possible de transférer

des informations confidentielles de la mémoire vers l'extérieur de la carte. Finalement, supposons qu'il soit possible pour une application de téléphonie mobile d'accéder aux ressources ou données d'une application bancaire, les données confidentielles de celle-ci pourraient être compromises. Ces exemples mettent en évidence le besoin d'établir des propriétés de sécurité afin d'exprimer de manière formalisée les besoins en sécurité d'un système embarqué. La confidentialité, l'intégrité, la bonne exécution ou le cloisonnement de certaines données pourront alors être vérifiés. Dans le chapitre 2, nous proposons une formalisation de certaines de ces propriétés de sécurité.

Les scénarii de création d'une vulnérabilité par l'injection d'une faute physique et le procédé permettant d'exploiter celle-ci pour obtenir un gain pour l'attaquant sont ce que nous nommerons *chemins d'attaque*. De nombreux chemins d'attaque existent suivant les tâches que doit accomplir le programme et l'implémentation qui en est faite. Dans le chapitre 5, nous utiliserons cette notion en nous plaçant dans un scénario fonctionnel donné et en cherchant exhaustivement par la simulation de fautes physiques à obtenir un gain fonctionnel.

1.3.4 Propriétés de sécurité

Les propriétés de sécurité, sur des systèmes embarqués, où la sécurité joue un rôle prédominant, doivent être exprimées en fonction d'aspects fonctionnels du système. En effet, une partie de la sécurité est intégrée dans des aspects fonctionnels du code. Une carte à puce a avant tout un but fonctionnel d'authentification. De cette fonctionnalité, on pourra ensuite tirer le gain à haut niveau pour l'attaquant, tel qu'il est décrit plus tard dans la section 1.3.7.

Une propriété de sécurité est une garantie souhaitée sur le code d'un point de vue de la sécurité. Elle est intrinsèque au système et peut être établie sans supposer d'attaque. À l'extrême, on peut envisager un attaquant avec des capacités nulles dans une modélisation qui fait apparaître un attaquant. Dans ce cas, la propriété de sécurité sera toujours garantie à moins que le développeur n'ait introduit une faille. Inversement, si un attaquant a des capacités infinies, on ne pourra jamais assurer la garantie d'une propriété de sécurité. La validation ou la mise en place des contre-mesures afin de garantir cette propriété dépend du modèle d'attaquant. Afin d'exprimer de manière pertinente une propriété de sécurité, il convient d'abord d'exprimer avec précision les capacités d'un attaquant à travers un modèle d'attaque. Nous établissons dans le chapitre 3 un tel modèle d'attaque.

Une autre considération à prendre en compte est la priorité de la fonctionnalité et des performances qui passe avant les aspects de sécurité du code. On pourrait dire que sécuriser un code qui ne réalise pas sa tâche principale ou qui n'atteint pas les performances requises est inutile car le produit, même s'il est parfaitement sécurisé, sera invendable.

Une dernière considération est le code en lui-même et la dynamique logique qu'il implémente. Le code exprime un besoin fonctionnel décrit sous forme d'une spécification fonctionnelle. Ainsi, les propriétés de sécurité, si elles sont garanties, protègeront ce besoin fonctionnel. Celui-ci correspond à une fonctionnalité implémentée dans un langage de programmation (ici le C) se décomposant en plusieurs éléments de ce langage : des fonctions et des variables.

Nous avons donc plusieurs éléments à prendre en compte pour exprimer les propriétés de sécurité.

- Le besoin fonctionnel sous-jacent ;
- L'implémentation logicielle ;
- Les capacités de l'attaquant.

Les spécificités d'une telle fonctionnalité dépendent totalement de ce que doit accomplir le programme. Il est donc difficile de définir des propriétés de sécurité génériques dans ce cadre. Cependant, une fonctionnalité se décompose en plusieurs éléments du langage de programmation comme une ou plusieurs fonctions et certainement une à plusieurs variables (locales et/ou globales). Des dépendances logiques comme des conditions et des boucles articulent dynamiquement ces éléments et donnent un sens au programme afin de réaliser le but de la fonctionnalité. Une propriété de sécurité cible une fonctionnalité que doit accomplir le programme. *Garantir une propriété de sécurité* au niveau du code source revient à valider que le code impliqué dans la sécurité d'une fonctionnalité résiste à des attaques. Une garantie de sécurité pour une fonctionnalité de haut niveau pourra être divisée en un ensemble de garanties sur des fonctionnalités d'un niveau inférieur plus proche du code source. Ainsi en vérifiant un ensemble de propriétés de sécurité sur le code source, la sécurité d'une fonctionnalité de haut niveau pourra être garantie.

1.3.5 Spécifications fonctionnelles et sécurités additionnelles

L'exemple du listing 1.1 montre un extrait de code implémenté à l'aide de spécifications fonctionnelles et de recommandations sécuritaires contre des attaques physiques. Cet exemple illustre la dualité de la sécurité : l'existence d'une sécurité fonctionnelle inclue dans les spécifications et d'une sécurité additionnelle rajoutée par le développeur contre les attaques physiques.

Soit la fonctionnalité d'authentification basique suivante :

- L'utilisateur doit s'authentifier à l'aide de son code PIN ;
- Si celui-ci est correct la carte doit renvoyer un jeton d'authentification au terminal ;
- L'utilisateur n'a droit qu'à 3 essais ;
- Le compteur d'essai doit être mémorisé en EEPROM.

Exemple de fonctionnalité sans sécurité additionnelle

La fonctionnalité décrite dans l'exemple précédent se décompose en quatre actions à accomplir dont une implémentation qui peut être la suivante.

Listing 1.1– Exemple d'une implémentation d'authentification basique avec une tâche fonctionnelle sécuritaire sensible mais sans code contre attaques physiques

```
void Authentification(void)
{
    user_pin = get_pin();
    card_pin = read_eeprom(&ee_card_pin);
    pin_cpt = read_eeprom(&ee_pin_cpt);
    if ((user_pin == card_pin) && (pin_cpt < 3))
    {
        return jeton_auth;
    }
    else
    {
        pin_cpt++;
    }
    write_eeprom(&ee_pin_cpt, pin_cpt);
}
```

Commentons comment ces actions ont été implémentées et ce que peut retirer l'attaquant en les perturbant :

Ligne 3 la valeur du PIN présenté par l'utilisateur est écrite dans la variable `user_pin` en RAM *Un attaquant peut modifier la valeur de la variable* RAM `user_pin` *mais n'en tire pas d'avantage car s'il possède la carte il est déjà capable de choisir la valeur de* `user_pin` *qu'il présente à la carte via le terminal*

Lignes 4 et 5 la valeur du PIN enregistré dans la carte est lue depuis l'EEPROM afin de la comparer à celle de l'utilisateur ; la valeur du compteur représentant le nombre de présentations successives invalides de la part de l'utilisateur est aussi lue ; ces valeurs sont respectivement écrites dans les variables en RAM `card_pin` et `pin_cpt`. *L'attaquant peut obtenir un avantage s'il modifie la valeur de* `card_pin` *afin qu'elle corresponde au PIN qu'il présentera à la carte. Par conséquent, cela lui permet de s'authentifier à la place de l'utilisateur légitime. Un autre gain pour l'attaquant peut être de forcer la lecture de* `ee_pin_cpt` *à une valeur inférieure à celle actuellement enregistrée en* EEPROM. *Cela lui permet, par un nombre illimité de présentations successives du PIN, de découvrir la valeur du PIN enregistré dans la carte.*

Lignes 6 à 13 le PIN présenté est comparé au PIN de la carte à la condition que le nombre d'essais incorrects successifs ne soit pas supérieur à 3 ; si ces conditions ne sont pas respectées, le compteur d'essai est incrémenté. *Au niveau de cette comparaison, un attaquant cherche soit à obtenir* `jeton_auth` *alors que la comparaison ne devait pas le diriger vers cette branche soit à empêcher l'incrément de* `pin_cpt` *afin de pouvoir, par un nombre illimité de présentations successives du PIN, découvrir la valeur du PIN enregistré dans la carte.*

Ligne 14 la valeur du compteur d'essais est mémorisée en EEPROM. *Ici aussi, l'attaquant cherche à perturber l'écriture afin que la valeur mémorisée en* EEPROM *soit inférieure à celle prévue lui permettant par un nombre illimité de présentations successives du PIN de découvrir la valeur du PIN enregistré dans la carte.*

Exemple de fonctionnalité avec sécurité additionnelle

Réaliser une attaque physique précise demande du matériel onéreux et une expertise. A cause de ces facteurs, un certain nombre d'hypothèses peuvent être faites sur ce qu'est capable de provoquer un attaquant. Ces hypothèses seront abordées dans le chapitre 3 et sont brièvement résumées ici afin de pouvoir comprendre les sécurités additionnelles à ajouter au code fonctionnel du listing 1.1. On suppose donc que l'attaquant :

- n'a droit qu'à une attaque physique au cours de la transaction ;
- peut modifier de manière transiente (non permanente) une variable en RAM lors de sa lecture (respectivement écriture) depuis (respectivement vers) l'EEPROM mais n'est pas capable de contrôler la valeur obtenue autre que `0x00` ou `0xFF` ;
- peut modifier de manière transiente la valeur d'une variable directement en RAM mais n'est pas capable de contrôler la valeur obtenue autre que `0x00` ou `0xFF` ;
- peut, sous certaines conditions, modifier de manière permanente une valeur lue depuis l'EEPROM ;
- peut, si la configuration matérielle et le contexte logiciel le permettent, effectuer deux fois la même attaque avec un certain écart temporel ;
- dans de très rares cas, effectuer des doubles attaques avec une modification spatiale de l'endroit visé ;

– peut sauter une ou plusieurs instructions successives en contrôlant l'origine du saut mais pas la destination dans le cas de plusieurs instructions successives.

Ces hypothèses se combinent avec la possibilité que possède l'attaquant d'arrêter le fonctionnement de la carte à n'importe quel moment en coupant l'alimentation de celle-ci. Un attaquant peut utiliser cette capacité pour empêcher le déclenchement des réponses sécuritaires, conservant ainsi la possibilité de réitérer son attaque. Il peut aussi l'utiliser pour perturber le flot d'exécution fonctionnelle du code en empêchant la réalisation de fonctionnalités ou l'écriture d'informations en EEPROM.

Ces différentes attaques dont nous avons brièvement rappelé les conséquences précédemment seront décrites en détail par la suite dans le chapitre 4. Dans le chapitre 4 nous nous intéressons aux conséquences des attaques physiques sur l'assembleur traduites au niveau du code C. Nous nous intéressons particulièrement aux attaques qui modifient la valeur des variables ou celles qui sautent du code.

Maintenant que nous avons décrit les possibilités de l'attaquant, nous allons donner une description des sécurités additionnelles permettant de se défendre contre certaines attaques. Nous ne couvrirons pas l'intégralité des contre-mesures mais seulement celles qui répondent aux attaques les plus plausibles car plus faciles à réaliser par l'attaquant.

Dans l'extrait de code 1.2, certaines sécurités additionnelles ont été rajoutées afin de sécuriser les fonctionnalités. Détaillons le rôle de chacune de ces sécurités :

Ligne 6 vérification de l'intégrité de `card_pin`, lue depuis l'EEPROM, en doublant sa lecture à l'aide d'une seconde variable `card_pin2`. En supposant que l'attaquant n'a droit qu'à une attaque perturbant une variable, il n'est pas capable de perturber à la fois la lecture EEPROM de `card_pin` et de `card_pin2`. La contre-mesure choisie est `kill_card()` (décrite dans la section 1.3.5). Cette contre-mesure n'opère cependant pas contre une double attaque ;

Ligne 13 vérification de l'intégrité de `pin_cpt` avec la même contre-mesure que pour `card_pin`. La réponse sécuritaire choisie ici est `kill_card()`. Le contenu de cette fonction est détaillé dans la section 1.3.5. Cette section contient aussi d'autres exemples de réponses sécuritaires ;

Lignes 19 et 22 on incrémente `pin_cpt` a priori pour éviter l'arrachement et on le décrémente si tout se passe bien, en supposant que l'incrémentation de `pin_cpt` à ligne 18 ne puisse pas être sautée en même temps que la condition à la ligne 20 ;

Ligne 28 vérification de l'intégrité de `pin_cpt` écrit en EEPROM en rechargeant en RAM la valeur écrite et en la comparant à celle supposée avoir été écrite. Cette contre-mesure suppose que l'attaquant ne peut réaliser qu'une seule attaque.

On peut remarquer que ces sécurités additionnelles dépendent fortement du contexte fonctionnel ainsi que des objets sensibles à protéger. Les lectures et écritures en EEPROM, si la donnée lue ou écrite est sensible (ici, `card_pin` et `pin_cpt`), sont particulièrement vulnérables aux attaques physiques et nécessitent une redondance sécuritaire. On note aussi que sécuriser du code rajoute de la complexité et augmente aussi sa taille tout en diminuant ses performances. Par exemple, la vérification de la cohérence de deux variables miroirs (ici, `card_pin2` et `pin_cpt2`) double au minimum le temps d'exécution.

Les organismes qui écrivent les spécifications des produits peuvent émettre des guides de recommandations sécuritaires confidentiels décrivant les objets sensibles et donnant des descrip-

Listing 1.2– Exemple d'une implémentation d'authentification basique avec une tache fonctionnelle sécuritaire sensible avec quelques défenses additionnelles contre les attaques physiques

```
void Authentification(void)                                                    1
{                                                                              2
    user_pin = get_pin();                                                      3
    card_pin = read_eeprom(&ee_card_pin);                                      4
    card_pin2 = read_eeprom(&ee_card_pin);                                     5
    // vérification de l'intégrité de card_pin                  (def 1)       6
    if (card_pin != card_pin2)                                                 7
    {                                                                          8
        killcard();                                                            9
    }                                                                          10
    pin_cpt = read_eeprom(&ee_pin_cpt);                                        11
    pin_cpt2 = read_eeprom(&ee_pin_cpt);                                       12
    // vérification de l'intégrité de pin_cpt                   (def 2)       13
    if (pin_cpt != pin_cpt2)                                                   14
    {                                                                          15
        killcard();                                                            16
    }                                                                          17
    pin_cpt++;                                                                 18
    // incrémentation a priori pour éviter l'arrachement        (def 3a)      19
    if (user_pin == card_pin) && (pin_cpt < 3)                                 20
    {                                                                          21
    // décrémentation si tout se passe bien                     (def 3b)      22
        pin_cpt--;                                                             23
        return jeton_auth;                                                     24
    }                                                                          25
    write_eeprom(&ee_pin_cpt, pin_cpt);                                        26
    re_ee_pin_cpt = read_eeprom(&ee_pin_cpt);                                  27
    // vérification de l'intégrité de pin_cpt écrit en EEPROM   (def 4)       28
    if (re_ee_pin_cpt != pin_cpt)                                              29
    {                                                                          30
        killcard();                                                            31
    }                                                                          32
}                                                                              33
```

tions haut niveau des contre-mesures qu'il serait recommandé d'implémenter au niveau fonctionnel. Les fondeurs peuvent aussi fournir ce genre de guide pour le développement de bas niveau décrivant comment implémenter par rapport à leur composant des contre-mesures efficaces. Ces dernières s'apparentent plus à des sécurités additionnelles.

Une des problématiques associée à cette double implémentation de la sécurité est le besoin de tester le bon fonctionnement de ces implémentations aussi bien dans le cas de la sécurité fonctionnelle que dans le cas de la sécurité contre attaque physique. Or, dans le cas de la sécurité fonctionnelle, il existe souvent des tests fonctionnels exigés par le client pour valider le comportement de la carte du point de vue du terminal. Cependant, dans le cas des contre-mesures contre attaque physique, le code rajouté dans le listing 1.2 ne devant pas modifier le comportement fonctionnel, il peut être considéré comme du code mort ne nécessitant pas de tests fonctionnels. Il faut alors créer des tests sécuritaires spécifiques pour stimuler et tester ce code rajouté.

Outre les tests spécifiques nécessaires pour valider le bon fonctionnement des contre-mesures implémentées, il existe une difficulté supplémentaire pour le développeur. Celui-ci doit identifier les éléments sensibles de son implémentation fonctionnelle et les sécuriser contre des attaques physiques potentielles. Or, le développeur n'a pas forcément une connaissance précise de l'impact d'une attaque physique sur le composant sur lequel son code sera déployé. Il lui est donc difficile d'estimer la sensibilité de son implémentation face aux attaques physiques.

Il est important de noter que, du point de vue du défenseur, un programme peut contenir du code que l'attaquant doit exécuter. Par exemple, le code correspondant à la mise à jour d'un compteur de sécurité ou à l'appel d'une réponse sécuritaire doit être exécuté, même en présence d'une attaque physique. Un programme peut, par ailleurs, contenir des parties de code que l'attaquant ne doit pas exécuter sous certaines conditions. Par exemple, un calcul cryptographique avec des données fixées par l'attaquant lors d'une attaque ne doit pas être effectué. Seul le développeur est capable de connaître le sens associé au code qu'il écrit et donc de définir les besoins en sécurité associés.

Sécuriser le code contre les attaques physiques est donc une tâche difficile pour le développeur qui est cependant le seul à disposer des connaissances nécessaires pour y arriver. Dans cette thèse, nous souhaitons cerner les possibilités d'attaque sur le code source et permettre au développeur de sécuriser plus facilement et plus efficacement son code source.

Contre-mesures sécuritaires

Plusieurs types de défenses existent pour se protéger contre les attaques physiques. Nous allons présenter celles-ci, le niveau auquel elles agissent et leur mode de fonctionnement.

Types de défenses Les défenses qui existent dans une carte à puce interviennent à plusieurs niveaux. L'empilement de ces défenses permet de créer plusieurs barrières de défense pour l'attaquant qui devra, par exemple, déjouer des défenses matérielles avant de pouvoir atteindre les parties du composant lui permettant d'influencer le comportement du programme. Ces défenses sont :

Matérielles où différents capteurs sont utilisés contre les différents vecteurs physiques existants (capteurs de lumière, de surtension, filtres, logique double rail ...). Ces défenses sont généralement transparentes pour le développeur. Des défenses matérielles supplémentaires

propres au composant peuvent exister mais demandent une configuration de la part du développeur.

Algorithmiques où l'implémentation par l'ajout de compteurs, de machines d'état et d'autres fonctionnalités assure un certain niveau de sécurité au système.

Logicielles cette partie est généralement laissée à la discrétion du développeur qui doit décider s'il est pertinent d'utiliser une redondance de données, doubler une opération, mettre en place un checksum ou des points de contrôle dans son code afin d'assurer la sécurité de celui-ci.

Détection de l'attaque Une défense peut généralement être décomposée en deux étapes. Premièrement, une vérification dans le code permet de détecter une incohérence par rapport au comportement attendu. Dans un second temps, une réponse sécuritaire est déclenchée en fonction de la sensibilité de l'opération en cours. Cette réponse peut être immédiate ou différée. Une détection dans une défense logicielle repose toujours sur une comparaison effectuée dans le langage de programmation. L'objet de la comparaison peut être multiple et dépend du contexte ainsi que de la sensibilité de l'objet à sécuriser ou de la fonctionnalité à sécuriser. Dans certains cas, les défenses algorithmiques et logicielles peuvent être imbriquées pour des gains de performance.

Dans le cas d'une fonctionnalité à sécuriser, la détection revient souvent à comparer le résultat obtenu à la fin de la fonctionnalité en exécutant une seconde fois l'intégralité ou une partie de la fonctionnalité. Par exemple, la mise à jour d'une donnée sensible comme un compteur en mémoire EEPROM demande une première mise à jour en RAM suivie d'une écriture en mémoire EEPROM puis d'une relecture et d'une comparaison par rapport à l'équivalent de la donnée présente en mémoire RAM.

Réponses sécuritaires Les réactions possibles face à une attaque sont souvent les suivantes :

- destruction de la carte par effacement de zones mémoires sensibles ;
- blocage d'une application qui se met en indisponibilité vis-à-vis du terminal ;
- dans certains cas, si le comportement face à l'attaque est spécifié, une réponse prévue par les spécifications peut être renvoyée au terminal.

Garantir une propriété de sécurité peut revenir à vérifier qu'une défense contre attaque physique existe, qu'elle couvre bien l'objet sensible à sécuriser, qu'elle se déclenche bien et provoque la réponse sécuritaire escomptée.

Positionnement d'une détection / contre-mesure Le positionnement d'un couple (détection et contre-mesure) dans le code doit être fait de telle manière que l'attaquant ne soit pas capable en une seule attaque d'obtenir un gain et de sauter la contre-mesure. Une autre contrainte dont le développeur doit tenir compte lors du positionnement est le fait que la carte peut être arrachée à tout moment. Ainsi, la défense doit être placée de telle sorte que l'attaquant ne puisse pas, par exemple, écrire une donnée sensible faussée en mémoire EEPROM, puis arracher la carte et ne pas effectuer la vérification, la contre-mesure ou les deux.

Portée d'une défense Sans contraintes de performance temporelles ou de taille mémoire, on aurait la possibilité de sécuriser une implémentation fonctionnelle sans se préoccuper des

conséquences. Cependant en embarqué, ces contraintes ne peuvent être ignorées. Il convient donc de les prendre en compte lors de l'implémentation de la sécurité en essayant d'optimiser, dans la mesure du possible, la portée des défenses. Une défense peut être qualifiée comme :

Effective sur une zone de code correspondant à portée de détection d'une défense couvrant ou non la fonctionnalité à sécuriser ;

Efficace si la fonctionnalité à sécuriser est couverte efficacement par cette portée ;

Optimale si la fonctionnalité à sécuriser est couverte efficacement par cette portée et que la portée minimise l'impact sur les performances.

L'objet à sécuriser est l'implémentation d'un besoin fonctionnel. La notion d'optimalité est liée à la durée de vie de l'objet à sécuriser et le type de propriété de sécurité qui s'y rapporte. On ne traite pas du coût mémoire d'une sécurité ni son optimisation dans cette thèse. Cependant, on s'intéressera à la couverture des attaques physiques par une défense logicielle en utilisant différentes techniques de simulation d'attaques. Quelle que soit la propriété de sécurité (confidentialité / intégrité), la durée de vie de l'objet à sécuriser aura son importance et sera liée à la définition de cette propriété de sécurité.

1.3.6 Sécurisation et angles d'attaque

Si on relie les besoins en sécurité au code source, on peut rapidement voir qu'il est difficile d'implémenter du code qui résiste aux attaques physiques. Prenons l'exemple d'une condition. Une version non sécurisé de l'implémentation d'une condition est présentée dans le listing 1.3.

Listing 1.3– Condition non sécurisée

```
if (x == y)
{
   z = 1;
}
```

Le listing 1.4 présente une première sécurisation consistant à rajouter une réponse sécuritaire si la condition n'est pas vérifiée.

Listing 1.4– Condition sécurisée naivement

```
if (x == y)
{
    z = 1;
}
else
{
    killcard();
}
```

Cette implémentation ne résiste pas à un attaquant capable de sauter l'exécution d'une instruction. En effet, prenons comme exemple une condition générique en C implémentée dans le listing 1.5. Celle-ci se traduit en assembleur par le code du listing 1.6 (exemple avec du code d'un composant Motorola 68000 dont on pourra trouver une description du jeu d'instructions à la référence suivante [EventHelix 2012]) :

Listing 1.5– Implémentation C d'une condition

```
if (x == y)
{
   z = 1;
}
else
{
   z = 0;
}
```

Listing 1.6– Traduction de cette condition en assembleur 68000

```
  * chargement de x dans le registre de donnée D7
  MOVE.L _x, D7
  * comparaison de x avec y
  CMP.L _y, D7
  * si inégalité saut dans la branche else spécifiée par le label L1
  BNE.S L1
  * chargement de 1 dans z
  MOVE.L #1, _z
  * saut après l'instruction if-then-else
  BRA L2
  * partie else de l'instruction if-else
  * mise à zéro de z
L1 CLR.L _z
  * fin de l'instruction if-else. En cas d'égalité, la partie else de la
  * condition est sautée pour atteindre directement le label L2
L2 ...
```

Or, dans le listing 1.6, si l'attaquant saute l'instruction à la ligne 6 ; même si la condition n'est pas vérifiée, le code de la branche `then` sera exécuté sans que celle de la branche `else` ne le soit jamais. Dans ce cas, un attaquant arrive à exécuter l'inverse de ce qui est prévu par le code (scénario 1).

On peut remarquer que si l'attaquant est capable de sauter l'instruction à la ligne 10, à partir du moment où il a accès à la branche `then`, il est en mesure d'exécuter à la fois la branche `then` et la branche `else`. On remarque aussi qu'en sautant les lignes 8 ou 13, l'attaquant peut ne pas exécuter le code effectif d'une des branches (car dans cet exemple, il n'y a qu'une instruction dans chacune des branches). Dans ce cas précis, cette attaque est équivalente au comportement initial ou à ne pas exécuter la totalité de la condition. Dans ces deux cas, l'attaquant doit être intéressé soit par l'exécution de la branche `else` (par exemple, si cette branche annule les effets de la première comme un compteur incrémenté et décrémenté) soit par la non exécution de l'ensemble de la condition (par exemple, le passage d'une machine d'état à un état désavantageux pour l'attaquant). Dans ces cas, l'attaquant arrive à exécuter plus ou moins que ce qui est prévu par le code (scénario 2).

Deux derniers cas intéressants à prendre en compte sont si l'attaquant saute une des instructions aux lignes 2 ou 4. Dans ces cas, l'attaquant empruntera une des branches non pas en fonction de la condition originale mais en fonction du contexte précédent le `if`. Dans le premier cas, la branche empruntée dépendra de la valeur se trouvant précédemment dans le registre $D7$ et de sa différence avec la valeur de y. Dans le second cas, la branche empruntée dépendra, avec cette architecture et jeu d'instruction, de la position du registre d'état Z qui est mis à 1 si le résultat de la soustraction entre les deux opérandes donne 0 [Angelis 2012]. Avec des jeux d'instructions différents, il pourra s'agir de l'`ACC`. Ce registre d'état peut être déterminé par d'autres

opérations, comme par exemple des opérations arithmétiques. La position de ce registre d'état définira la branche empruntée après l'exécution de l'instruction de saut à la ligne 10. Dans ces cas, l'attaquant arrive, en fonction du contexte, à obtenir un comportement identique au comportement original, un comportement inverse ou un de ces comportements avec les effets de bord induits par la non exécution d'une des lignes sautées (scénario 3).

Une sécurisation possible contre l'attaque du scénario 1 est l'implémentation sécurisée du listing 1.7. En supposant que la fonction `killcard()` termine l'exécution du programme sans retourner dans le code qui l'appelle, la branche `else` ne sera jamais exécutée.

Listing 1.7– Condition sécurisée avec une première méthode

```
1 if (x != y)
2 {
3     killcard();
4 }
5 else
6 {
7     z = 1;
8 }
```

Cependant, cette forme de code défensive ne sécurise pas contre les scénarii d'attaque 2 et 3. Elle ne fonctionne pas non plus contre des sauts de multiples instructions consécutives. Par exemple, ne pas exécuter l'instruction d'appel à la fonction `killcard()` et le saut à la ligne 10 peut avantager l'attaquant si celui-ci cherche à exécuter le contenu de la branche `else` alors que la condition est fausse (`x != y`). Cette défense ne protège pas non plus contre les doubles attaques qui permettent, par exemple, le saut du contenu d'une branche (ligne 8) et le saut d'une instruction de branchement (ligne 10), ce qui, dans certains cas, peut être équivalent au saut de multiples instructions successives. Les doubles sauts sont cependant plus efficaces dans l'hypothèse d'une modification de code qui avantage l'attaquant suivi du saut de la contre-mesure associée.

La seule défense possible qui prend en compte les scénarii 1, 2 et 3 (sans pour autant prendre en compte les doubles sauts et certains sauts d'un grand nombre d'instructions successives) est de doubler la condition (si possible avec une implémentation différente de la première) et de vérifier la cohérence entre ces deux conditions. Cependant, cette méthode double au minimum le temps d'exécution ainsi que la mémoire nécessaire pour implémenter la condition.

Ce travail de définition du gain pour l'attaquant, d'analyse des effets des fautes sur le comportement du code et de lien entre l'effet et le gain devrait être fait sur l'ensemble des formes de code du système et une version défensive de cette forme de code devrait être utilisée lors d'une opération sensible. Dans le chapitre 5, nous proposons une méthode exhaustive afin de tester dynamiquement les comportements possibles du code en injectant exhaustivement des attaques au niveau C mais aussi au niveau assembleur. Cette méthode permet d'évaluer la résistance des contre-mesures implémentées en réduisant au maximum le travail de définition et d'analyse.

1.3.7 Modélisation et vérification

Dans un premier temps, afin de pouvoir proposer des méthodes cherchant à garantir la sécurité de ces systèmes embarqués, nous proposons de formaliser des propriétés de sécurité exprimant les garanties de sécurité souhaitées. Dans un second temps, afin de pouvoir cerner l'ensemble des problématiques liées à la sécurité embarquée, il convient d'établir un modèle des

capacités de l'attaquant, c'est-à-dire, sa capacité à modifier le comportement fonctionnel du code par le biais d'une attaque physique. Grâce à ces éléments, nous proposerons des outils permettant l'analyse du code et la vérification de la sécurité de celui-ci.

Modélisation de la sécurité : une approche fonctionnelle

Une approche possible pour modéliser la sécurité est l'approche fonctionnelle. Les problématiques de sécurité sont abordées à partir des fonctionnalités que doit accomplir le programme. L'avantage d'une telle approche est qu'elle reste générique par rapport au code et s'adapte à un environnement industriel. Un logiciel a toujours un cahier des charges décrivant ses fonctionnalités. Une fois les tâches à accomplir par le programme connues, on peut facilement en déduire le gain pour un attaquant. Par exemple, si une tâche est d'authentifier un utilisateur s'il présente un bon PIN, le gain pour l'attaquant est de s'authentifier avec un mauvais PIN. De plus, on peut aisément, en lisant le cahier des charges, isoler les fonctionnalités sensibles sur lesquelles se concentrer et éliminer celles dont la sécurité est moins importante. Par conséquent, une représentation de la sécurité à ce niveau est adaptée à un contexte industriel par son caractère générique. En effet, on arrivera toujours à exprimer un besoin de sécurité à partir du cahier des charges du logiciel.

Finalité fonctionnelle Soit une transaction regroupant plusieurs commandes. Chaque commande peut être vue comme une fonctionnalité disposant d'une finalité. Cette finalité est généralement présente dans la réponse renvoyée par la carte mais aussi dans les changements opérés au sein de la carte par l'exécution de la commande. L'ensemble de ces commandes mène à la finalité globale qui est l'aboutissement de la transaction représentant la tâche fonctionnelle globale. En faisant l'hypothèse que la dernière réponse de la carte reflète l'exécution d'un ensemble d'opérations internes relatives à la transaction, on peut en vérifiant cette réponse déduire que ces opérations internes se sont bien effectuées. Cette hypothèse est à la base des tests en boîte noires. Cette hypothèse revient aussi à dire que l'attaquant ne peut pas à la fois réaliser une attaque et perturber le mécanisme de gestion d'erreur et aussi qu'une gestion d'erreur est implémentée pour chaque opération critique au sein de la carte. La réponse finale à l'échelle d'une transaction, reflète le déroulement d'opérations issues de commandes précédentes car dans une transaction (par exemple EMV) ces commandes sont liées par des informations partagées. Pour notre modélisation nous prendrons une perspective à l'échelle d'une commande. Les propriétés de sécurité peuvent être établies sur des commandes ou à l'échelle d'une transaction.

Perspective et vision allégée Si on analyse le flot de contrôle du code source, on peut distinguer :

- des opérations ou calculs ;
- des appels de fonctions regroupant un ou plusieurs opérations et calculs. Ces fonctions peuvent être vues comme des boites avec des entrées et sorties ;
- des conditions.

Le listing 1.8 illustre ce niveau d'abstraction en ne faisant apparaître que les deux derniers éléments mentionnés ci-dessus. Ainsi, une fois que le terminal a envoyé une requête à la carte, le code présent dans celle-ci s'exécute de manière autonome sans autre intervention de la part du terminal. Les seules informations modifiant les décisions lors de l'évaluation des conditions

Listing 1.8– Requête schématique

```
# Terminal --envoi d'une requête--> Carte
# -in->
F()
{
    if (toto ==  titi)
    {
        f();
    }
}
# <-out-
```

du code se trouvent dans la requête d'origine. En gardant une représentation faisant seulement apparaître les éléments du graphe d'appels et les éléments du flot de contrôle, nous avons abstrait le code présent qui s'exécute séquentiellement pour ne garder que les points de changement possibles dans la logique du programme. Ces points de décision sont clés dans la logique de fonctionnement du programme et représentent des points d'attaques particulièrement intéressants pour un attaquant.

Ces visions de l'exécution fonctionnelle permettent focaliser uniquement sur un sous ensemble du graphe d'appels total, i.e., les séquences d'appels se terminant par une finalité fonctionnelle. Les transitions à l'intérieur de ces séquences (conditions, appels) sont des points dans le code potentiellement vulnérables à une attaque physique car ce sont des points clés au niveau fonctionnel.

D'un point de vue de la sécurité, on peut considérer qu'une vérification, par exemple doubler une opération comme dans le listing 1.9, déclenchera en cas de non respect une réponse sécuritaire et n'amènera jamais à appeler la fonction permettant de continuer dans le graphe d'appels vers la finalité. En effet, doubler une opération n'apporte rien dans le flot fonctionnel du code mais est uniquement présent pour protéger contre des attaques physiques.

Dans l'exemple 1.9, après une vérification de l'exécution correcte de `calcul_crypto()`, le flot du code retournera sur le chemin fonctionnel standard et effectuera une transition normale hors de la fonction `F()`. Par contre, si la condition `if (res1 == res2)` n'est pas vérifiée, le flot d'exécution standard est interrompu et bascule sur l'exécution du code d'une réponse sécuritaire.

Cette perspective nous montre qu'il est important de considérer un point de vue particulier lorsqu'on s'intéresse à la sécurité fonctionnelle et que seule une partie du code sera pertinente dans la modélisation.

Expression du gain pour l'attaquant

Le gain pour l'attaquant est lié à l'utilisation faite de la carte à puce. Une carte à puce peut, par exemple, être utilisée dans le cadre d'une authentification entre plusieurs partis. Dans ce scénario, un attaquant dispose de plusieurs moyens pour exploiter les possibilités d'une attaque physique. Nous décrivons dans la suite deux types d'exploitation suivant que l'attaque compromet des données ou modifie le comportement de la carte.

Si on considère qu'une attaque permet d'obtenir des informations ou des données confidentielles permettant d'authentifier les partis (typiquement des clés d'authentification), le but de l'attaquant est de se faire passer pour un des partis afin de se placer comme destinataire d'un

Listing 1.9– Exemple simple d'une fonction avec doublement sécuritaire qui n'a pas d'impact sur la finalité fonctionnelle

```
// on arrive fonctionnellement ici
// appel à F

void F(void)
{
    res1 = calcul_crypto();

    // de peur d'être attaqué on double le calcul
    res2 = doublement_calcul_crypto();

    if (res1 == res2)
    {
        // transition sur le chemin fonctionnel
        continuer_fonctionnellement_vers_finalite();
    }
    else
    {
        // non transition
        killcard();
    }
}
```

transfert financier et intercepter le montant transféré. Dans la pratique, l'attaquant fabrique des clones de la carte contenant les données confidentielles obtenues malicieusement. Dans le cas d'un vol de carte, l'attaquant cherche en priorité à obtenir le code PIN du possesseur de la carte ce qui lui permettra de retirer de l'argent.

Si on considère qu'une attaque physique est capable de modifier le comportement fonctionnel de la carte, le but de l'attaquant est de générer de l'argent (en ciblant les compteurs de montant), de monter en privilège dans des scénarii qui ne le demandent pas (en ciblant des variables d'état), de faire du rejeu de commande (en ciblant des variables de machines d'état), d'empêcher le verrouillage sécuritaire de la carte (en empêchant la mise à jour de variables d'état sécuritaire). Dans la pratique, l'attaquant a besoin de l'assistance d'un commerçant complice afin de réussir son attaque dont le gain monétaire n'est pas immédiatement visible.

Cependant modifier le comportement du code peut amener à une exploitation indirecte dans un contexte différent. Par exemple, le rejeu autorise la génération de traces exploitables dans des attaques par canaux cachés afin de retrouver des clés secrètes. On retombe dans un des cas mentionnés ci-dessus où l'attaquant cherche à se faire passer pour l'un des partis. Le rejeu inconditionnel d'une commande ouvre aussi la possibilité d'épuiser exhaustivement les combinaisons d'entrée et ainsi découvrir un secret.

On remarque ainsi qu'il existe de nombreux vecteurs d'attaques qui dépendent à la fois du contexte d'utilisation de la carte et de son implémentation fonctionnelle.

Afin d'assurer une complétude au niveau de la sécurité fonctionnelle et se prémunir contre un maximum de vecteurs d'attaque possibles, il convient de réaliser un travail de réflexion préliminaire sur les besoins fonctionnels et l'implémentation de ceux-ci dans une optique de sécurité.

1.3.8 Angle d'approche et résumé

Dans cette thèse, nous cherchons à améliorer la sécurité de la carte en introduisant une meilleure visibilité des effets des fautes physiques. Nous adoptons une approche fonctionnelle de haut niveau afin de mettre en évidence comment ces fautes peuvent être utilisées pour monter des attaques sur le code. Afin de répondre à cette problématique de sécurité, nous allons modifier le comportement d'un programme en faisant varier certains paramètres le caractérisant.

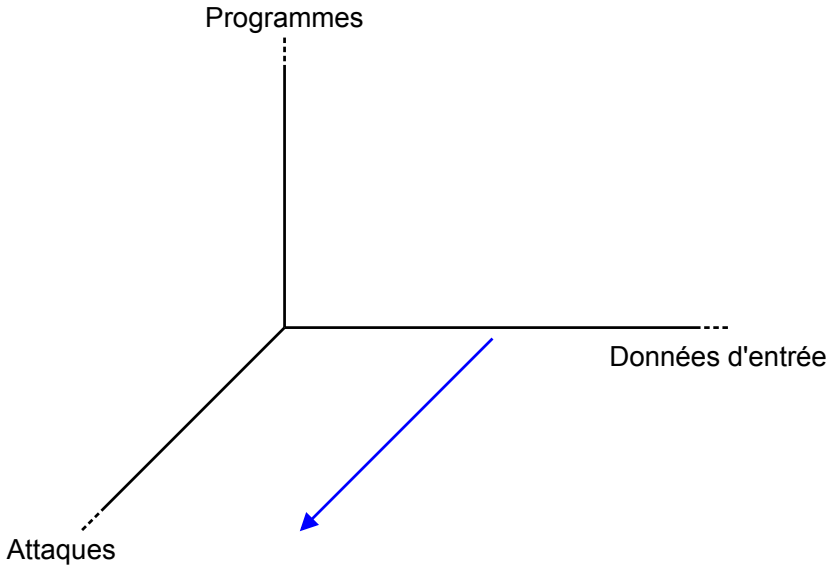


FIGURE 1.14 – Axes d'approche du problème

La figure 1.14 montre une perspective visuelle du problème et l'angle d'approche que nous souhaitons adopter. Les différents axes représentent les paramètres du problème qui peuvent être modifiés. Dans notre approche, nous ferons varier les attaques possibles en fixant le code d'un programme et des données d'entrée spécifiques. Des approches différentes comme le fuzzing [Lancia 2011] ne prennent pas en compte d'attaques et font varier les données d'entrée sur un même programme. Une méthode utilisant la mutation de programme [Barbu *et al.* 2010] essaie de couvrir l'ensemble des programmes possibles dont ceux contenant une attaque pour un même jeu d'entrées. Nous avons choisi cette méthode qui semble dans notre contexte être la plus appropriée. Dans cette thèse, nous exploiterons donc cette méthode consistant à faire varier les attaques possibles sur l'ensemble du programme.

En plus du choix de cette méthode, nous adoptons également un point de vue particulier dans cette thèse.

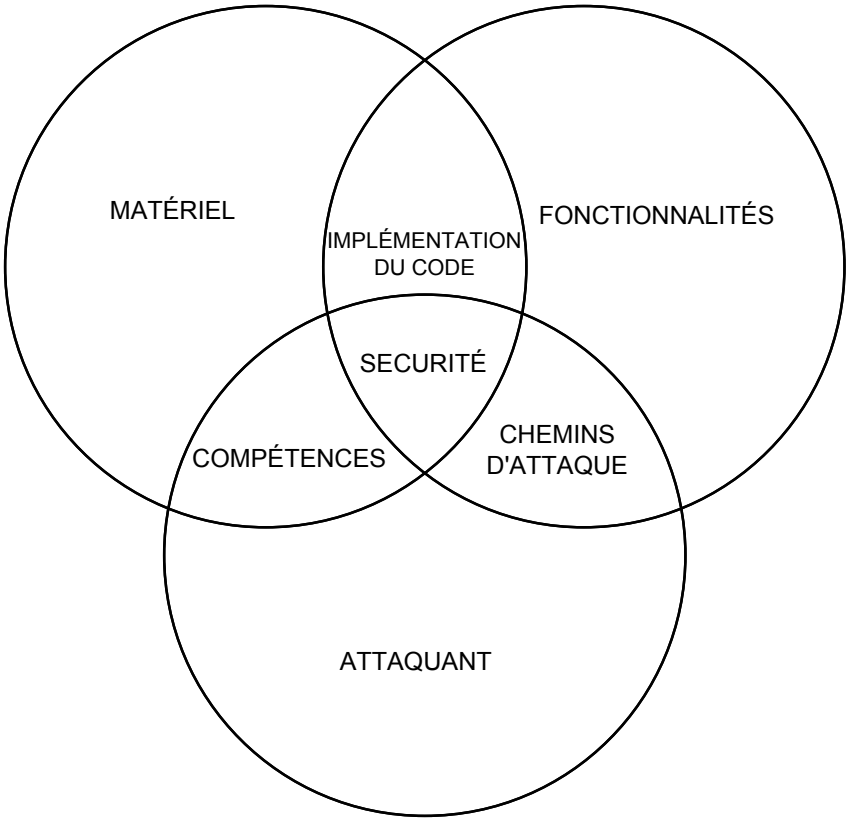


FIGURE 1.15 – La place de la sécurité dans le contexte embarqué

La sécurité sur une carte à puce se trouve à la jonction de plusieurs éléments répartis à différents niveaux :

Matériel le circuit électronique responsable de l'exécution physique du programme ;

Fonctionnel la logique derrière le fonctionnement du matériel définissant le ou les rôles et actions à accomplir ;

Attaquant la personne malveillante dont le gain va s'exprimer en fonction du fonctionnel.

Ces éléments sont eux mêmes interdépendants :

Implémentation logicielle entre le matériel et le fonctionnel, représente comment le développeur va traduire les actions fonctionnelles en un programme s'exécutant sur le matériel (la puce) ;

Les chemins d'attaques chemins logiques représentant comment l'attaquant envisage de troubler l'exécution fonctionnelle afin d'obtenir un gain ;

Les compétences matérielles de l'attaquant définissent le matériel auquel l'attaquant a accès et ses connaissances techniques lui permettant de mener à bien une attaque.

A chaque niveau peut être associé un point de vue particulier qui englobe les considérations et problématiques associées au niveau en question. L'approche adoptée dans cette thèse est de prendre en compte le point de vue du développeur chargé d'implémenter le code fonctionnel et d'incorporer une notion d'attaquant matériel directement traduite sur le code source.

1.4 Conclusion et annonce

Dans ce chapitre, nous avons présenté l'architecture d'un microcontrôleur afin de mieux cerner les vulnérabilités en cas d'attaques physiques. Nous avons aussi donné l'intuition nécessaire pour comprendre ce que cherchait à réaliser un attaquant grâce à ces attaques. Les composants matériels de l'architecture d'un microcontrôleur ont été passés en revue et nous avons présenté l'aspect fonctionnel des considérations en sécurité. Nous avons mis en évidence le besoin d'exprimer des garanties de sécurité sous forme de propriétés de sécurité génériques au niveau du code source du programme. Nous avons également mis en évidence le besoin d'un modèle d'attaque au niveau du code source permettant de formaliser les capacités de modification fonctionnelle d'un attaquant à l'aide d'une attaque physique. Finalement, nous avons montré qu'une méthode permettant de tester le code source contre ces attaques physiques était nécessaire. Dans les chapitres suivants, nous présenterons donc nos quatre principales contributions qui sont regroupés dans les publications suivantes : [Chambérot & Kauffmann-Tourkestansky 2009], [Berthomé *et al.* 2010], [Berthomé *et al.* 2011] et [Berthomé *et al.* 2012].

Dans le chapitre 2, une formalisation de propriétés de sécurité au niveau du code source est établie. Des propriétés de sécurité comme la confidentialité, l'intégrité de données et l'intégrité d'exécution sont formalisées de manière à ce qu'un développeur soit capable de les utiliser directement sur son code source.

Dans le chapitre 3, un modèle d'attaque pour la vérification de propriétés de sécurité est établi sur le code C. Ce modèle d'attaque permet de déterminer une méthode pour simuler les capacités d'un attaquant sur le code source. Nous déterminons notamment ses capacités à observer et à agir sur les composants matériels d'un microcontrôleur et les répercussions sur le code source. Dans ce chapitre, nous introduisons également une modélisation statistique du phénomène de décalage du flot d'exécution résultant d'une attaque physique.

Dans le chapitre 4, ce modèle d'attaque établi est utilisé pour réduire le nombre d'attaques à réaliser. Nous apportons la preuve que cette réduction stimule l'intégralité des attaques possibles du modèle proposé ainsi qu'une technique déterminant les points d'attaque de cette réduction. Nous utilisons également une approche statique visant à montrer qu'il est possible d'utiliser une méthode d'injection de fautes simulées et de la relier à une méthode de vérification formelle.

Dans le chapitre 5, nous proposons une évaluation des risques de sécurité liés aux attaques physiques sur le contrôle de flot. Cette évaluation est réalisée grâce à une simulation à haut niveau de ces attaques utilisant le modèle précédemment créé. Ce chapitre établit un classement de ces attaques en catégories pertinentes vis-à-vis de leur conséquence fonctionnelle. Nous cherchons ainsi à relier les attaques matérielles et le comportement fonctionnel résultant par la simulation d'attaques au niveau logiciel. Cette relation est utilisée pour montrer les conséquences sur le contrôle de flot du programme en simulant des attaques par saut. La méthode proposée permet de mettre en évidence les conséquences fonctionnelles quelle que soit l'attaque par saut. Une approche dynamique est ensuite utilisée visant à déterminer le degré de couverture des attaques possibles au niveau physique par le modèle établi et les simulations effectuées à haut niveau. Cette couverture est établie en utilisant différentes techniques d'injection d'attaque.

Le chapitre 6, décrit différents outils existants pour effectuer des tests ou des vérifications de la sécurité dans des projets à échelle industrielle. Ces outils seront passés en revue et nous montrons le besoin d'une solution dédiée répondant à un contexte précis. Les différents outils industriels réalisés dans le cadre de cette thèse sont présentés en détail et nous mettrons en lu-

mière leurs avantages dans l'adoption d'une méthodologie complète et adaptée à l'environnement industriel de cette thèse.

Finalement, le chapitre 7 terminera cette thèse en présentant les conclusions établies et les perspectives possibles du travail réalisé.

Chapitre 2

Modélisation et propriétés de sécurité

Sommaire

2.1 Introduction, approche et hypothèses 49
2.1.1 Approches existantes 50
2.1.2 Décomposition de fonctionnalité 53
2.1.3 Décomposition en éléments du langage 53
2.2 Formalisme utilisé 55
2.2.1 Information et conteneur 55
2.2.2 Ensemble d'informations visibles 56
2.2.3 Zone de garantie 61
2.2.4 Type d'attaques 61
2.2.5 Formalisation d'une propriété de sécurité 61
2.3 Confidentialité d'une information 62
2.3.1 Confidentialité sous attaque 64
2.3.2 Zone de génération d'information d'un conteneur 64
2.3.3 Zone de garantie de confidentialité 66
2.3.4 Avantage pour l'attaquant et granularité de l'information 67
2.4 Intégrité 69
2.4.1 Accès à un conteneur et zone de garantie 69
2.4.2 Intégrité en lecture et écriture 70
2.4.3 Avantage pour l'attaquant 73
2.4.4 Intégrité d'exécution 73
2.5 Conclusion 75
2.6 Perspectives 77
2.6.1 Aspects sémantiques 77
2.6.2 Temporalité des propriétés 77
2.6.3 Prise en compte du compilateur 77
2.6.4 Théorie de l'information 78
2.6.5 Génération automatique de propriétés 78

2.1 Introduction, approche et hypothèses

Après avoir montré dans le chapitre précédent, le besoin de formaliser la sécurité afin de cerner les besoins en sécurité d'un code embarqué. Nous avons pris le parti à la fois d'adopter une vision fonctionnelle permettant d'appréhender directement le gain pour l'attaquant et un point de vue proche du développeur afin que celui-ci soit capable d'exprimer les besoins en sécurité de son code. Nous exprimons ainsi des propriétés de sécurité sur le code source utilisé

par le développeur, ici le C. Ces propriétés doivent aussi être génériques afin de pouvoir être utilisées quel que soit le projet.

Dans ce chapitre, nous parlerons donc d'une modélisation de la sécurité permettant de formaliser des propriétés de sécurité. Ces propriétés de sécurité nous servent à vérifier des comportements de sécurité sur notre système. Dans notre approche, nous allons choisir de les modéliser à partir d'un point de vue particulier à mi-chemin entre des spécifications fonctionnelles de haut niveau et des éléments du langage de programmation. Une caractéristique originale dans cette approche est l'incorporation de réponses sécuritaires dans la formalisation. La particularité propre au contexte de la sécurité embarquée est que le matériel peut être attaqué physiquement. Cette particularité intervient lors de la vérification et du test de ces propriétés.

Dans ce chapitre, nous nous intéressons donc à la définition de propriétés de sécurité dans un environnement embarqué soumis à des attaques physiques. Dans un contexte industriel, le choix des composants utilisés et de la chaine de compilation est laissée à l'entreprise développant la partie logicielle de la carte. Ces caractéristiques peuvent varier d'un projet à l'autre et excluent l'utilisation du compilateur lors de la vérification des propriétés de sécurité. Ces caractéristiques rendent aussi la définition de propriétés génériques plus difficiles. Notre objectif est de formaliser des propriétés de sécurité pour les raisons données dans la section 1.3 du chapitre 1. Ce chapitre a permis de donner l'intuition de ce que pouvait faire un attaquant. Nous voulons maintenant formaliser les garanties de sécurité souhaitées à l'aide d'un modèle et des propriétés de sécurité.

Pour formaliser les garanties de sécurité souhaitées, nous devons d'abord nous intéresser à ce que fait le code puis à ce que l'attaquant gagnerait à le perturber. Les fonctionnalités du programme une fois décomposées en opérations proches du langage de programmation utilisé, peuvent être projetées sur le code source.

2.1.1 Approches existantes

Différentes approches à la définition de propriétés de sécurité existent dans la littérature. Ces approches adoptent des points de vue différents entraînant des niveaux d'abstraction plus ou moins éloignés du code source.

SELinux [McCarty 2004] est un exemple de modèle de sécurité qui peut être rajouté au système standard de Linux afin d'augmenter la sécurité face aux diverses attaques que subit le système. Ce modèle centré sur le contrôle d'accès adopte une approche de haut niveau et permet, par l'utilisation de rôles et de permissions, de garantir la sécurité. Ce modèle a été déployé avec succès sur des systèmes embarqués Android afin de garantir des propriétés de légitimité d'accès à des ressources [Shabtai *et al.* 2010].

Des modèles créés pour assurer des garanties d'isolation entre différentes parties d'un même système existent également. Les auteurs de [Schellhorn *et al.* 2002] proposent une extension des modèles de Bell/Lapaluda [Bell & LaPadula 1975] et Biba [Biba 1977] afin de vérifier formellement la sécurité d'une carte multi-applicatives dans le cas où une nouvelle application est ajoutée. Les deux derniers modèles cités définissent la confidentialité et l'intégrité en terme de *contrôle d'accès* dans un système avec des acteurs possédant des degrés de privilège différents. Le point de vue adopté est porté sur le droit d'accès à des ressources du système d'exploitation se traduisant par des lectures et écritures dans des fichiers du système de fichier. Ce point de vue s'intéresse au système et reste relativement éloigné du code source.

Schwan mentionne dans [Schwan 2008], en plus des modèles précédents, des modèles basés sur

le *flot d'information*. Originalement introduit par Denning [Denning 1976], ce modèle a donné lieu à des prédicats comme la *non-interférence* [Goguen & Meseguer 1982]. Ce prédicat définit des règles pour le transfert d'une information sensible entre des sources d'un niveau de sécurité différent. Schwan mentionne d'autres prédicats comme la restriction, la non-déductibilité et l'isolation. Ces prédicats sont formalisés à l'aide de *traces*. Une trace étant une caractérisation du système réalisée à l'aide d'une séquence d'entrées et de sorties. Le point de vue adopté confronte des menaces à des objectifs de sécurité formalisés et regroupés en politiques de sécurité. Cette approche adaptée à un travail de certification reste relativement éloignée du code source. Cependant, la notion de flot d'information et de trace se rapproche de celui-ci. En effet, les transferts d'information sont visibles au niveau du code source tandis que les traces d'exécution caractérisent un scénario fonctionnel. Ce scénario peut être lié à l'approche fonctionnelle présentée dans la section 1.3.7 du chapitre 1.

Le concept de trace est utilisé par Thorn [Thorn 1999] dans une approche de la sécurité adoptant un point de vue proche du langage de programmation. Celui- ci choisit une représentation en terme de graphe du programme. Une trace d'exécution du programme projetée sur cette représentation génère une séquence de nœuds qui peut être confrontée à des propriétés de sécurité. Ces propriétés de sécurité sont définies à partir de formules logiques qui font intervenir des séquences de nœuds préétablies et labellisés suivant des critères de sécurité. Ces critères de sécurité illustrent la validité ou le potentiel de risque d'une séquence d'appels du programme. Cette approche est appliquée au code d'une Java Card et valide l'utilisation du graphe d'appel d'un programme pour valider des propriétés de sécurité portant sur une suite d'appels du programme. Cette approche a l'avantage de permettre, une fois les séquences d'appels connus, d'établir des règles qui pourront être validées par la machine virtuelle de manière statique ou dynamique. Elle demande cependant de définir à l'avance ces séquences d'appels. Ce modèle ne suppose pas non plus la présence de fautes à l'intérieur du code d'un appel. Si certains aspects de cette approche peuvent être adaptés au niveau C, la majeure partie reste spécifique à un environnement utilisant le langage Java.

Les auteurs de [Sabelfeld & Myers 2003] donnent une description complète des prédicats associés au flot d'information utilisé pour la vérification de la confidentialité. Des détails sur l'utilisation de l'analyse statique pour la vérification de politiques de confidentialité ainsi qu'une présentation des défis associés sont donnés. On note que l'auteur effectue un lien entre la confidentialité et l'intégrité : la confidentialité requiert qu'une information ne soit pas transmise dans des destinations incorrectes tandis que l'intégrité demande que des informations ne soient pas transmises depuis des sources incorrectes. Si ces deux définitions ne prennent ni la temporalité des transferts ni la persistance des informations en compte, elles donnent une idée des propriétés de sécurité au niveau du langage où les sources et destinations sont des variables. Un programme, en calculant des valeurs erronées, peut corrompre l'intégrité d'informations sans influence extérieure. Une vérification complète de l'intégrité est donc plus difficile à mettre en œuvre car elle demande une preuve de correction de l'intégralité du programme. Les auteurs mettent aussi l'accent sur la difficulté à vérifier les flots d'information sur des langages de bas niveau car la structure de celui-ci est perdu à la compilation. L'analyse statique et l'interprétation abstraite [Cousot & Cousot 1977] sont cependant considérées prometteuses pour la vérification de flots d'information dans un contexte de sécurité.

Les preuves formelles constituent aussi un moyen de garantir la sécurité. Les auteurs de [Andronick *et al.* 2005] utilisent un système à base d'annotations, spécifiant formellement

les fonctionnalités du programme en terme de préconditions et postconditions. Ces fonctionnalités couplées à une représentation formelle du programme extraite du code source forment un modèle de haut niveau sur lequel des propriétés de sécurité peuvent être vérifiés à l'aide d'outils de preuve. Les auteurs utilisent ces système et modèle afin de vérifier le mécanisme de journalisation du système d'exploitation. Leur approche produit une garantie forte du fonctionnement du mécanisme et donc de la sécurité des opérations d'écriture en EEPROM associées. Les auteurs de [Blazy & Leroy 2005] proposent une approche similaire mais utilisent un modèle de la mémoire à mi-chemin entre la représentation mémoire C et une représentation mémoire matérielle. Ces derniers donnent en plus des indications sur le temps d'exécution nécessaire à l'exécution des générations et vérifications de preuves. Aucune indication sur le coût en temps nécessaire à la définition de ces preuves n'est donnée. Les résultats de [von Oheimb *et al.* 2003] laissent cependant penser que de telles preuves peuvent passer à l'échelle dans des contextes industriels. L'auteur de [Lanet 2000] émet quelques réserves sur l'utilisation de méthodes formelles pour tout un projet dans un cadre industriel car le coût nécessaire à leur mise en place dans un processus existant est important. La définition des spécifications formelles ainsi que la connaissance des outils de preuve nécessaires rend difficile cette mise en place. Une utilisation limitée à des parties d'un système peut cependant, sous certaines conditions, être praticable.

Un exemple de propriété de sécurité, exprimée formellement directement sur le code source, est le *flot d'information sécurisé*. Ce flot d'information sécurisé est défini dans le modèle établi par Denning comme une structure en treillis. Cette structure symbolise spatialement les niveaux de sécurité présents dans un système. L'auteur de [Zanotti 2002], en utilisant ce modèle, donne la définition suivante de ce treillis : un ensemble partiellement ordonné $(SC, \leq)$, avec C un ensemble fini de niveaux de sécurité muni de la relation d'ordre partiel $\leq$ (aussi appelé relation de dominance). Chaque variable x utilisée à l'intérieur du programme possède un niveau de sécurité $\underline{x}$. Il est supposé que ce niveau de sécurité est déterminé statiquement et qu'il ne peut changer à l'exécution. La définition suivante est établie :

Définition 1 (Flot d'information sécurisé) *Si x_1 et x_2 sont deux variables du programme et qu'il existe un flot d'information de x_1 vers x_2, le flot peut être autorisé si et seulement si $\underline{x_1} \leq \underline{x_2}$*

Avec cette définition, l'hypothèse fixant le niveau de sécurité à l'exécution entraine que des cas particuliers ne peuvent être pris en considération.

Listing 2.1– Cas particulier

```
int generique[2] = {1234,0};     // un tableau avec une information          1
                                 // sensible 1234 et une non sensible 0      2
int apdu;                        // variable qui transmet des informations   3
                                 // à l'extérieur de la carte                4
apdu = generique[0];             // affectation qui pose un problème         5
                                 // de sécurité                              6
envoie_terminal( apdu );         // fonction qui divulgue l'information      7
```

Dans le listing 2.1, on ne peut pas associer de niveau de sécurité au tableau `generique` qui contient des informations avec des niveaux de sécurité antagonistes. La variable `apdu`, à laquelle on peut affecter un niveau de sécurité bas, pose un problème de sécurité si elle est la cible d'une affectation d'une information sensible. Avec la définition précédente, il n'est pas possible de déterminer si cette affectation est autorisée ou pas. De plus, supposons que l'affectation est

`apdu = generique[1];`. Celle-ci devrait être autorisée. Cependant, en cas d'attaque physique, l'effet matériel provoqué peut perturber le code interprété de telle manière à retrouver `apdu = generique[0];`. Les attaques physiques posent donc des problèmes de sécurité supplémentaires.

Dans ce chapitre, nous donnons des définitions de propriétés de sécurité qui prennent en compte les problématiques exposées ci-dessus en introduisant des notions supplémentaires. Ces définitions s'inspirent de concepts mentionnés dans les contributions de cette section. Elles essaient de répondre à la problématique d'exprimer des notions de sécurité sur un langage de programmation alors que celui-ci peut, sous attaque, ne pas avoir le comportement souhaité.

2.1.2 Décomposition de fonctionnalité

La sécurité d'une fonctionnalité du programme, telle qu'elle est décrite dans les spécifications fonctionnelles, est fortement liée à l'avantage qu'un attaquant pourrait tirer du fait de réussir son attaque. Nous avons défini dans le chapitre précédent, dans la section 1.3.7, ce que nous appelons le gain pour l'attaquant. Le modèle de sécurité est construit sur l'étude des gains potentiels de l'attaquant lorsqu'il réalise des attaques contre une fonctionnalité du code.

Sans attaque, les propriétés de sécurité seraient naturellement garanties puisque le code fonctionnel est censé, par construction, respecter les propriétés de sécurité voulues. Dans ce cas, des contre-mesures additionnelles ne seraient pas nécessaires. En cas d'attaque, une réponse sécuritaire doit être déclenchée si une propriété de sécurité n'est pas respectée. Il est possible qu'une attaque n'ait pas d'effet ou que l'effet ne perturbe pas assez l'exécution pour qu'une propriété de sécurité soit violée. Cependant, dans le cas contraire, une contre-mesure de sécurité doit contrecarrer l'attaque. Les contre-mesures sont donc placées à titre préventif dans le code. Celles-ci ne doivent pas empiéter sur le fonctionnel et changer le comportement attendu. Dans le chapitre suivant, nous développons la notion d'attaquant, dans cette section nous présentons à travers un exemple comment exprimer les besoins de sécurité sur le code à partir des spécifications fonctionnelles.

Listing 2.2– Exemple d'une implémentation basique d'un mécanisme d'authentification

```
void auth(void)
{
    pin = get_pin();
    if (pin == 1234)
        return jeton_auth;
}
```

Dans l'exemple du listing 2.2, implémentant une fonction d'authentification basique, un gain pour l'attaquant serait d'obtenir le jeton d'authentification avec le mauvais PIN (ou sans vérification du PIN). L'objectif fonctionnel de la fonction `auth` est décrite dans les spécifications du programme. Pour exprimer le besoin en sécurité de cette fonctionnalité, l'expert en sécurité doit donc s'intéresser aux spécifications et en déduire le ou les gains pour l'attaquant. Les besoins en sécurité peuvent alors s'exprimer en fonction de l'inverse de ces gains.

2.1.3 Décomposition en éléments du langage

Une fonctionnalité s'implémente à l'aide d'informations et de logique. Cela se traduit, au niveau du code source, par des éléments *conteneurs* comme des variables ou des tableaux et des éléments *logiques* comme des conditions (`if`) ou des répétitions (`for`, `while`) assemblés pour

remplir l'objectif d'une fonctionnalité (l'incrément d'un compteur et sa mise à jour en mémoire EEPROM, par exemple).

L'implémentation d'une fonctionnalité peut faire intervenir du code source à différents endroits du programme. Les éléments *conteneurs* et *logiques* sont alors répartis dans différentes fonctions. De plus, dans l'ensemble du code implémentant une fonctionnalité, seul un sous-ensemble de celui-ci peut avoir du sens en terme de sécurité. L'expert en sécurité et le développeur doivent donc s'associer pour déterminer le code qui doit être protégé.

Les points du programme ainsi définis représentent les points d'attaque principaux du programme par rapport à une fonctionnalité. Suivant l'implémentation du programme, des points d'attaque indirects peuvent être trouvés. Si une condition est un point d'attaque, les affectations des variables impliquées dans cette condition sont des points d'attaque indirects. L'ensemble de ces points d'attaque aussi appelés *chemin d'attaque* doivent être pris en compte afin de complètement sécuriser une fonctionnalité. Les opérations associées à ces points dans le programme peuvent alors être caractérisées. Ainsi, une propriété de sécurité peut se traduire en *contraintes* sur des éléments du langage. Soit des besoins de sécurité de haut niveau comme par exemple :

- cette condition doit être exécutée ;
- ce calcul doit être réalisé ;
- à ce point dans l'exécution du programme, pour ce scénario, cette variable doit avoir une telle valeur ;
- cette action doit être effectuée avant telle action.

Ces besoins vont engendrer des contraintes sur les éléments du langage.

- pour exécuter le code de la branche `then` d'un `if`, la condition impliquant l'égalité entre deux variables doit être vraie ;
- cette variable locale doit être incrémentée ;
- cette variable globale de la machine d'état doit être mise à jour à cet endroit du programme ;
- l'appel à cette fonction doit être réalisé seulement si cette variable a été vérifiée a priori.

Dans un environnement sans attaque, ces contraintes sont naturellement garanties, ce qui n'est pas le cas en présence d'attaque. On a ici l'intuition que, s'il existe une dépendance d'information entre deux variables du programme, cette dépendance est relayée en matière de sécurité à l'intérieur du programme. Cette dépendance met aussi en évidence le lien de conséquence entre les *informations* abritées dans des éléments *conteneurs* et les éléments *logiques* du programme. Une relation de conséquence inverse existe aussi. Ces liens sont à la base des *chemins d'attaque* indirects et sont donc importants pour la sécurité. Une dernière notion visible ici est l'équivalence des conséquences entre plusieurs points d'attaque. En effet, empêcher l'exécution d'une des branches d'une condition revient à sauter l'exécution du code correspondant à la branche ou à fausser l'évaluation de la condition pour qu'elle exécute l'autre branche ou encore à fausser un terme de la condition pour que la branche souhaitée soit exécutée. Il existe donc des chemins d'attaque plus ou moins directs permettant d'atteindre et d'exploiter une vulnérabilité fonctionnelle. Ces notions sont exploitées dans la suite de cette thèse.

Pour plus de précision, on définit une propriété de sécurité à un niveau proche d'une opération unitaire dans le code source. Ainsi, afin de sécuriser entièrement une fonctionnalité, qui peut contenir un nombre conséquent d'opérations mémoires et logiques, il convient de décomposer celle-ci en autant d'opérations unitaires à sécuriser et donc d'instances de propriétés de sécurité à garantir afin d'assurer la sécurité globale de la fonctionnalité.

2.2 Formalisme utilisé

2.2.1 Information et conteneur

La notion d'*information* et celle de *conteneur* sont importantes pour la compréhension des propriétés de sécurité. Un programme embarqué sur une puce est une suite d'instructions qui modifie et transforme des informations en réponse à des informations de la part du terminal. Ces informations transitent à travers des conteneurs définis au niveau du langage de programmation. Ainsi, on peut définir la notion de conteneur au niveau du langage de programmation.

Définition 2 (Conteneur et Information) *Un conteneur C contient une information I à un instant t. Un conteneur est un élément physique dans le système carte à puce - terminal.*

Soit PIN un objet fonctionnel, on note I_{pin} l'information associée à cet objet et C_{pin} un conteneur abritant cette information.

Différents cas de figure sont envisageables lorsque ces notions sont projetées sur le code source. Une variable, à une ligne donnée du code source, permet d'accéder à un emplacement physique qui contient une information. Cette information prend la forme d'une valeur numérique. À l'exécution du programme, cette information est modifiée, transmise, dupliquée, effacée. Cependant, elle reste véhiculée de conteneur à conteneur.

Listing 2.3– Transfert d'information par conteneur

```
int apdu;                                    1
int pin = 1234;                              2
apdu = 0000;                                 3
apdu = pin;                                  4
```

Dans le listing 2.3, le conteneur C_{apdu} de l'objet nommé *apdu* correspond à la variable `apdu` dans le code. L'information abritée par ce conteneur est I_{apdu} qui a pour valeur `0000` puis `1234` après *transfert d'information* de la variable `pin` à la ligne 4. On remarque que cette information est transmise au conteneur `apdu` sans être effacée du conteneur `pin`.

Un conteneur abrite, à un instant précis, une valeur. Cette valeur est son information par défaut. Cette valeur peut cependant coïncider, de manière fortuite ou non, avec la valeur d'informations abritées par d'autres conteneurs. Ainsi, à la ligne 4, l'information I_{apdu} du conteneur `apdu` est `1234` qui est aussi, à cause de la duplication de l'information, l'information I_{pin} du conteneur C_{pin}. Donc après l'exécution de cette ligne de code, on a $I_{apdu} = I_{pin}$.

Listing 2.4– Évolution d'une information par affectation directe de valeur

```
int pin;                                     1
pin = 0000;                                  2
pin = 1234;                                  3
```

La valeur d'une information peut aussi évoluer par *affectation directe* sans provenir d'un autre conteneur. Dans le listing ci-dessus, C_{pin} = `pin` avec I_{pin} = `0000`. Après l'affectation de la ligne 3, on a C_{pin} = `pin` avec I_{pin} = `1234`.

Ainsi, à un instant donné, un objet fonctionnel peut se trouver réparti grâce à son information dans différents conteneurs. On peut aussi remarquer qu'au niveau du code source, le nom d'une variable n'est pas représentatif d'un objet fonctionnel qui est seulement caractérisé par une information.

2.2.2 Ensemble d'informations visibles

Grâce à la notion de conteneur et d'information, nous pouvons maintenant décrire ce qu'est une information observable par un attaquant et aussi dans quel cas la confidentialité de cette information est respectée ou compromise. Nous cherchons ici à caractériser la *divulgation d'une information* au travers de conteneur.

Le code du listing 2.5 permet d'illustrer les différentes notions définies dans cette section. La figure 2.1 présente les transferts d'information lors de l'exécution de ce code et leur visibilité en fonction de la puissance d'observation d'une attaquant.

Listing 2.5– Transfert d'information sans attaque

```
int var1, var2, pin, clef, mem, apdu;
mem = 88888;
pin = 1234;
var1 = pin;
var2 = var1;
var2 = mem;
apdu = var2;
print_terminal(apdu);
clef = 536237;
var1 = clef;
```

La figure 2.2 montre les différents points de vue que peut avoir un attaquant et les informations qui lui sont accessibles. Chacun de ces points de vue est caractérisé par l'ensemble des conteneurs qu'un attaquant est en mesure d'observer. Ces derniers sont définis en fonction des capacités de l'attaquant. La couleur utilisée reflète la sensibilité de l'information abritée par un conteneur : rouge et rose pour sensible et vert pour non sensible. La figure permet de visualiser l'évolution des informations à l'intérieur du code. Dans cette figure, en se plaçant au niveau du terminal, les informations transitant par les conteneurs C_{apdu} sont visibles. Avec des moyens d'observation plus avancés, comme l'observation des canaux cachés, un attaquant peut aussi avoir accès aux informations contenues dans C_{var1} ou C_{var2}. Avec une sonde physique, un attaquant observe les informations transitant par un conteneur mémoire C_{mem}. Les conteneurs C_{pin} et C_{clef} (et donc les informations qu'ils peuvent contenir) restent invisibles à un quelconque observateur.

Dans ce scénario, si l'attaquant se place du point de vue du terminal, il peut prendre connaissance des informations du conteneur `apdu` ainsi que celles des variables `var2` et `mem`, par transfert d'information. L'information `1234`, initialement enregistrée dans la variable `pin`, reste confidentielle car l'attaquant est incapable d'observer, depuis le terminal, le contenu des variables `pin` ou `var1`.

Un critère important est le moment d'observation de l'attaquant. Ainsi, si un attaquant a accès, par canaux cachés, aux informations abritées par le conteneur `var1` à l'instant *t1*, il a accès à l'information sensible `1234`. Cependant, s'il observe ce même conteneur à *t2*, il a accès à une autre information sensible `536237`. Une défense contre les canaux cachés peut donc être de réduire le champ de vision de l'attaquant au moment où des données sensibles sont manipulées en mémoire ou à rendre une exécution sensible non différentiable d'une exécution non sensible.

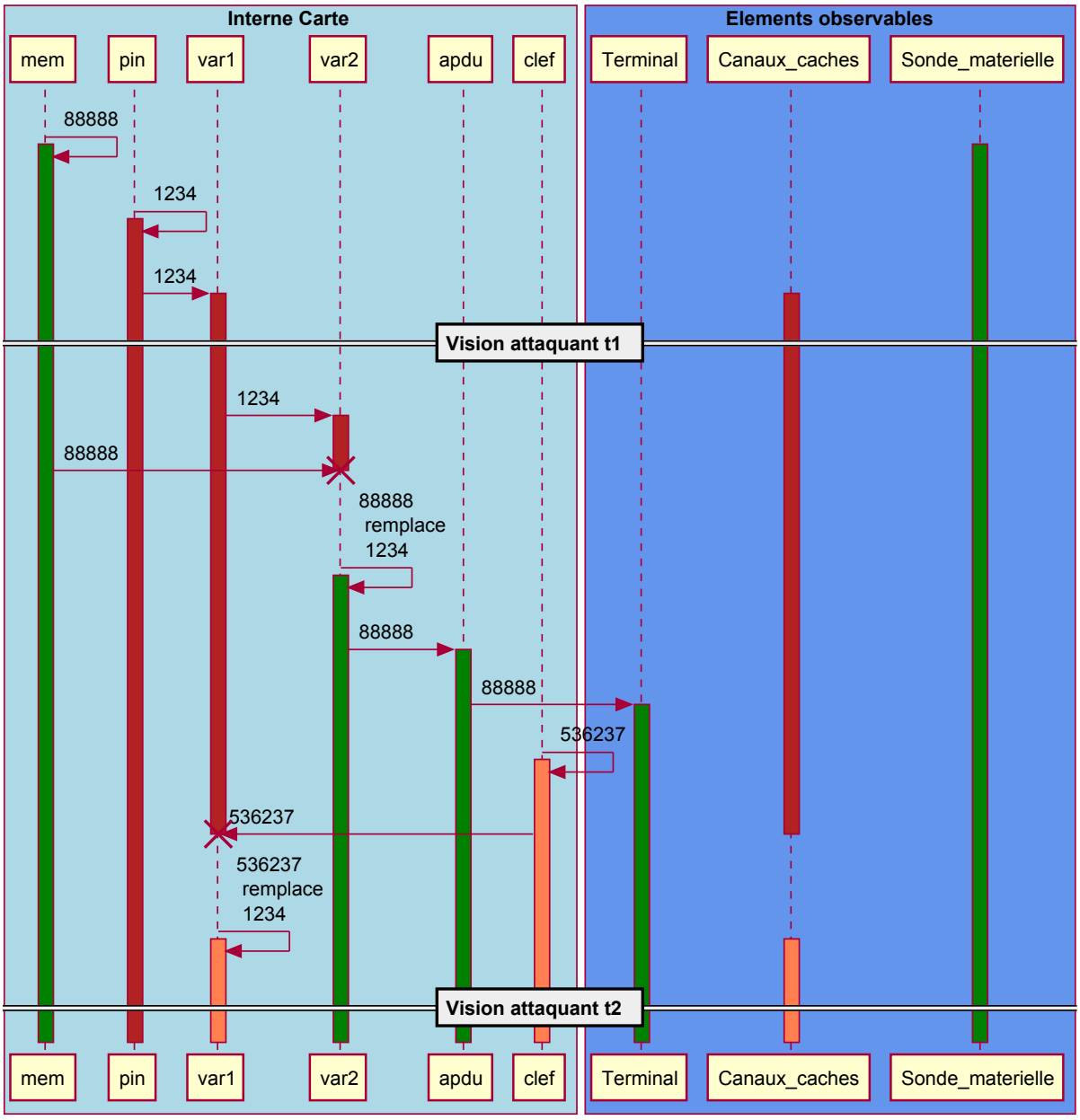


FIGURE 2.1 – Évolution des informations à l'intérieur des conteneurs correspondant au code du listing 2.5

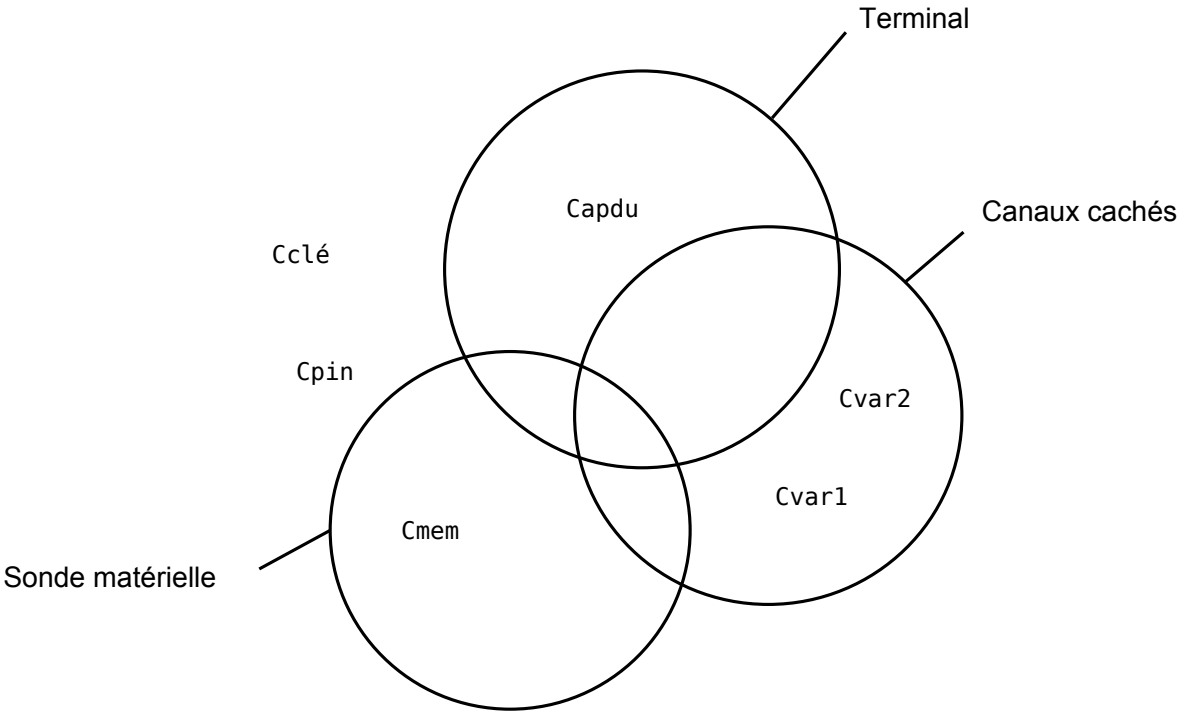


FIGURE 2.2 – Avec une méthode d'observation et un point d'observation certains conteneurs, ainsi que l'information qu'ils abritent, sont visibles

Forte capacité d'observation Il est intéressant de remarquer que suivant les compétences d'observation de l'attaquant, certaines informations peuvent lui être directement accessibles. Ainsi, si l'attaquant peut, par observation, avoir constamment accès aux informations abritées par le conteneur PIN, il est capable de compromettre la confidentialité de n'importe quelle information sensible transférée à l'aide de ce conteneur. La figure 2.3 illustre cet exemple, dans le cas où l'attaquant est capable, par canaux cachés d'avoir constamment accès aux informations transitant par le conteneur C_{pin}. Ainsi, si le code du listing 2.5 est exécuté, l'attaquant a directement accès à l'information `1234`, sans même avoir besoin d'effectuer une attaque physique modifiant le flot d'information.

Un moyen de se protéger contre un attaquant de ce niveau est de chiffrer l'information avant son passage dans le conteneur ; ce qui n'est pas toujours possible. Se protéger contre les attaques par canaux cachés est donc une tâche ardue, où caractériser précisément l'attaquant est nécessaire mais difficile. Les défenses implémentées doivent prendre en compte une surapproximation des capacités de l'attaquant pour ne pas être mise en défaut.

Attaque physique modifiant le flot d'information En cas d'attaque, le flot d'information peut changer et de nouveaux transferts d'information peuvent être créées. Le listing 2.6 reprend l'exemple précédent en simulant la non exécution de l'instruction `var2 = mem;` ligne 6. Ce type d'attaque correspond, par exemple, à une attaque par saut d'instruction.

Une telle attaque peut introduire une vulnérabilité provoquant la perte de confidentialité d'une information sensible. La figure 2.4 montre l'évolution des informations à l'intérieur du

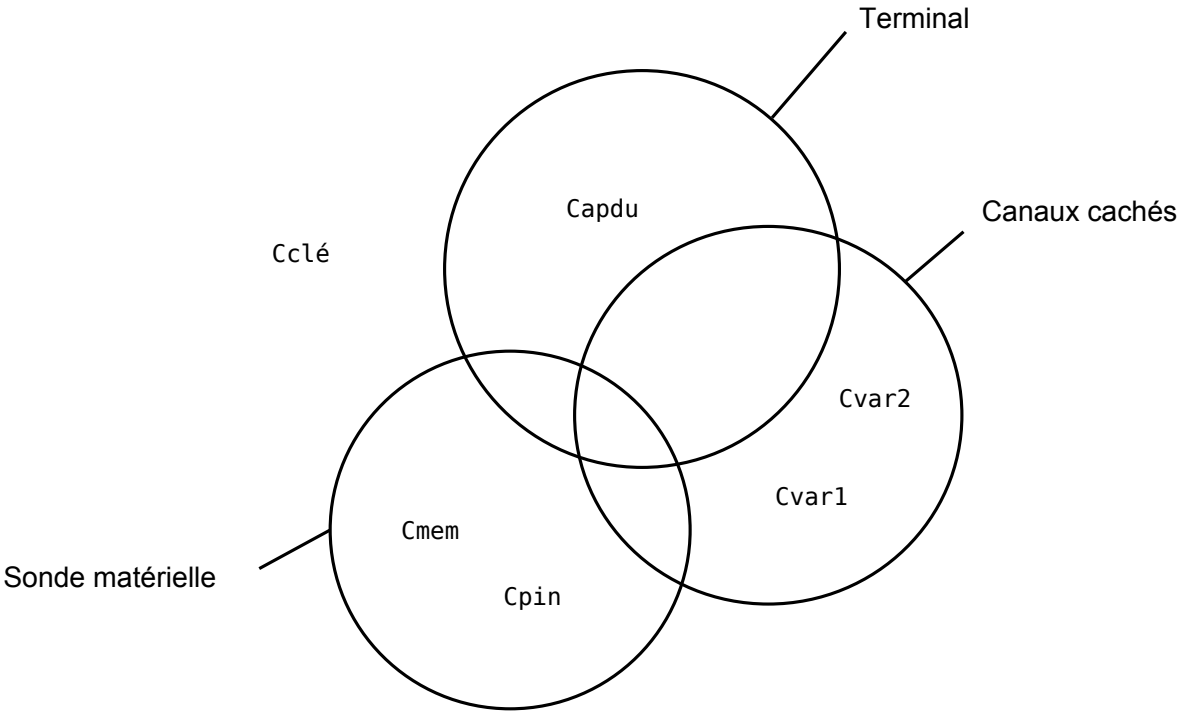


FIGURE 2.3 – La confidentialité est directement compromise si une information se trouve dans un conteneur visible

Listing 2.6– Transfert d'information avec attaque

```
int var1, var2, pin, cle, mem, apdu;
mem = 88888;
pin = 1234;
var1 = pin;
var2 = var1;
var2 = mem;
apdu = var2;
print_terminal(apdu);
clef = 536237;
var1 = clef;
```

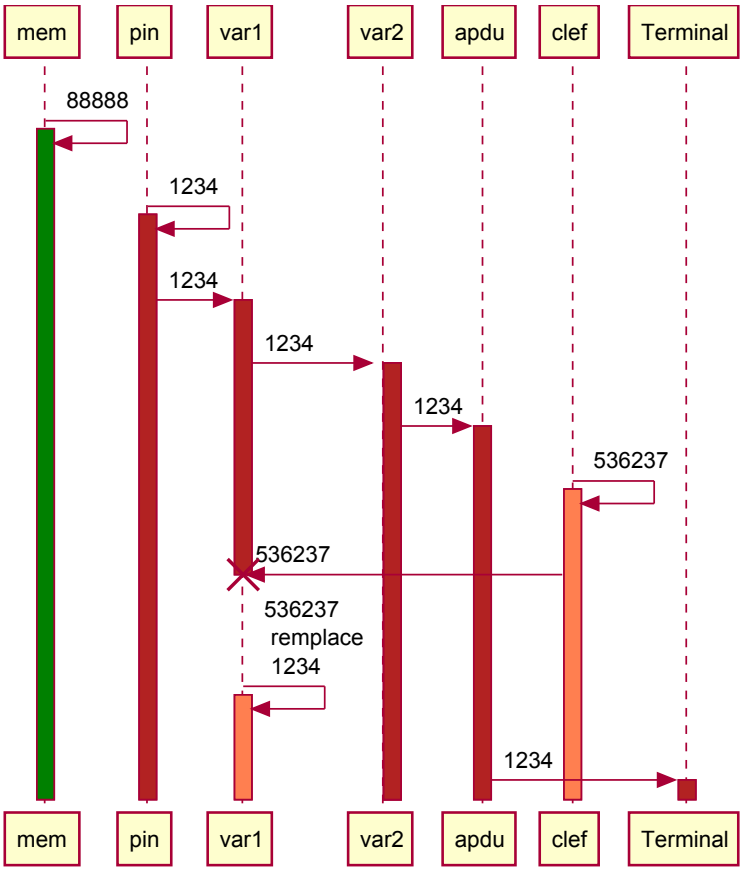


FIGURE 2.4 – Évolution des informations à l'intérieur des conteneurs correspondant au code du listing 2.6

code du listing 2.6. On observe que l'information sensible I_{pin} valant `1234` se trouve maintenant dans un conteneur visible par l'attaquant C_{apdu} alors que ce n'était pas le cas. Le saut d'une instruction a ainsi changé le flot d'information au travers des variables jusqu'à communiquer une information sensible, qui devait rester confidentielle, à l'extérieur de la carte.

Dans un contexte où le système peut être soumis à des attaques physiques, nous devons considérer, en plus des capacités d'observation de l'attaquant, ses capacités à agir sur le système. Ainsi, nous introduisons $\mathcal{T}_A$, le type d'attaque, symbolisant les capacités d'action de l'attaquant, autrement dit, comment il peut réussir à perturber l'exécution du code. Un exemple de capacité d'action est la possibilité d'introduire des sauts dans le code mentionné précédemment. Si $\mathcal{T}_A = \varnothing$, alors l'attaquant a la capacité d'action d'un utilisateur standard par le biais du terminal.

Définition 3 (Ensemble d'éléments observables sous attaque) *L'ensemble des informations obtenues en observant le conteneur C dans l'hypothèse d'une attaque de type $\mathcal{T}_A$ est noté $Obs(C, \mathcal{T}_A)$.*

Pour violer la confidentialité d'une information secrète, un attaquant cherche à découvrir directement l'information par canaux cachés en analysant les fuites d'information générées lors

de l'exécution du programme. Il peut aussi provoquer une faute physique afin de faire dérailler le flot d'information et amener des informations sensibles dans des conteneurs visibles depuis l'extérieur. Ainsi, la combinaison des capacités d'observation et d'action de l'attaquant peut lui permettre de compromettre la sécurité du système.

2.2.3 Zone de garantie

En plus des notions d'information et de conteneur, une notion de zone doit être définie pour restreindre le champ d'application d'une garantie souhaitée. En effet, sachant que plusieurs informations peuvent être contenues à des instants différents dans un même conteneur, il convient de borner l'application d'une propriété de sécurité à une zone particulière où elle a du sens.

Définition 4 (Zone de garantie) *Une zone de garantie Z_G est une zone de code correspondant à un contexte temporel dans lequel le respect d'une propriété de sécurité est demandée par la garantie de sécurité.*

Par exemple, on cherche à garantir que le PIN reste confidentiel pendant la commande `Verify_PIN()` mais pas pendant la commande `Update_Record()`. Cette zone de garantie se traduit en une zone de code dans une ou plusieurs fonctions C où on cherche à vérifier la propriété pour cette information.

2.2.4 Type d'attaques

Les attaques sont des éléments perturbateurs qui vont modifier le comportement du système à l'exécution. Différents types d'attaques sont possibles en fonction des capacités de l'attaquant. Les différents types d'attaques sont développés dans le chapitre 3 et regroupée dans un *modèle d'attaque*. Pour l'instant, nous supposons que les attaques se décomposent en deux types : celles sur les informations manipulées par le programme et celles contre le programme manipulant ces informations. Comme mentionné précédemment, une attaque peut modifier le flot d'information et amener des informations secrètes dans le champ de vision de l'attaquant. Cependant, elle peut aussi directement compromettre l'intégrité de certaines données ou opérations, lui permettant ainsi d'influencer les décisions prises à l'exécution.

2.2.5 Formalisation d'une propriété de sécurité

On peut maintenant exprimer de manière générique une propriété de sécurité en faisant apparaître les principales composantes de celle-ci.

Définition 5 *Soit la notation suivante, pour une propriété de sécurité P :*

$$P(x, a, Z_G, code)$$

avec

- *P : la propriété ;*
- *x : des éléments sensibles ou actifs à protéger ;*
- *a : un ensemble d'attaques suivant un modèle d'attaque ;*
- *Z_G : une zone de garantie pour la propriété de sécurité ;*

– *code : le code source sur lequel s'applique la propriété (sous entendu par la suite).*

Les éléments sensibles x à protéger sont des éléments du programme qui ont un rôle fonctionnel important dans la réalisation de tâches sensibles. À chacun de ces éléments, il est possible d'associer une garantie de sécurité souhaitée (confidentialité, intégrité). Cette garantie doit être respectée à un moment de l'exécution du programme symbolisé ici par la notion de zone de garantie Z_G. On veut, par exemple, assurer que le PIN reste confidentiel lors de la commande `Verify_PIN()`, que le compteur d'essai du PIN reste intègre lors de la même commande ou qu'une clé reste confidentielle et intègre lors de son utilisation dans un calcul cryptographique.

Rappelons qu'un attaquant se modélise en deux capacités distinctes : sa capacité à observer le système et sa capacité à agir sur ce système. La formalisation des propriétés de sécurité prend en compte ces deux capacités qui seront détaillées par la suite dans le chapitre 3. La capacité d'observation rajoute une composante à la propriété, pour la confidentialité, tandis que la capacité à agir est symbolisée ici par a.

Les éléments décrits ci-dessus sont les principales composantes constituant des propriétés de sécurité que nous déclinons dans les prochaines sections de ce chapitre. Nous formalisons ainsi deux besoins de sécurité : la confidentialité et l'intégrité.

2.3 Confidentialité d'une information

Pour cette propriété, un conteneur est défini comme un ou plusieurs éléments matériels (case mémoire RAM ou EEPROM, registre) d'un système embarqué pouvant abriter à un moment donné une information. En projetant cette représentation sur le C, les variables sont donc des conteneurs labellisés. On note C un conteneur.

On souhaite définir la notion de confidentialité sur le code source d'un projet. Or, dans un code source, la notion d'information est difficilement représentable directement. Passer par un conteneur abritant cette information permet de s'affranchir de cette difficulté. Cela permet aussi au développeur de spécifier dans le code un conteneur et d'induire à partir de ce conteneur les informations qu'il convient de protéger en confidentialité.

Une information a une durée de vie dans un conteneur. Cette durée de vie débute quand l'information est introduite dans le conteneur et se termine lorsque une nouvelle information prend sa place. Cette durée de vie d'une information à l'intérieur d'un conteneur ne doit pas être confondue avec la durée de vie du conteneur, définie comme la durée pendant laquelle ce conteneur est accessible. Au sens C, c'est-à-dire en adoptant le point de vue du programme, une variable locale est un exemple de conteneur dont la durée de vie est équivalente à la fonction où elle est définie.

Le listing 2.7 donne un exemple de transfert d'information entre plusieurs conteneurs basé sur l'exemple du listing 2.5. Nous nous servons de cet exemple pour illustrer les notions de durée de vie d'un conteneur et de durée d'une information dans un conteneur.

Dans ce code, la durée de vie des conteneurs `var1`, `var2`, `pin`, `clef`, `mem`, `apdu`, des variables locales, est confinée à la fonction `foo()`. Un conteneur global a une durée de vie débutant au moment de sa création en mémoire et terminant quand il n'est plus possible d'accéder à la mémoire lui étant dédié. La durée de vie d'une information à l'intérieur d'un conteneur est différente. Par exemple, la durée de vie de l'information `1234` à l'intérieur du conteneur `var2` débute en ligne 7 et se termine en ligne 8. Cet exemple est représenté dans la figure 2.5.

Listing 2.7– Exemple de transfert d'information dans des conteneurs

```
void foo(void)
{
    int var1, var2, pin, clef, mem, apdu;
    mem = 88888;
    pin = 1234;
    var1 = pin;
    var2 = var1;
    var2 = mem;
    apdu = var2;
    print_terminal(apdu);
    clef = 536237;
    var1 = clef;
}
```

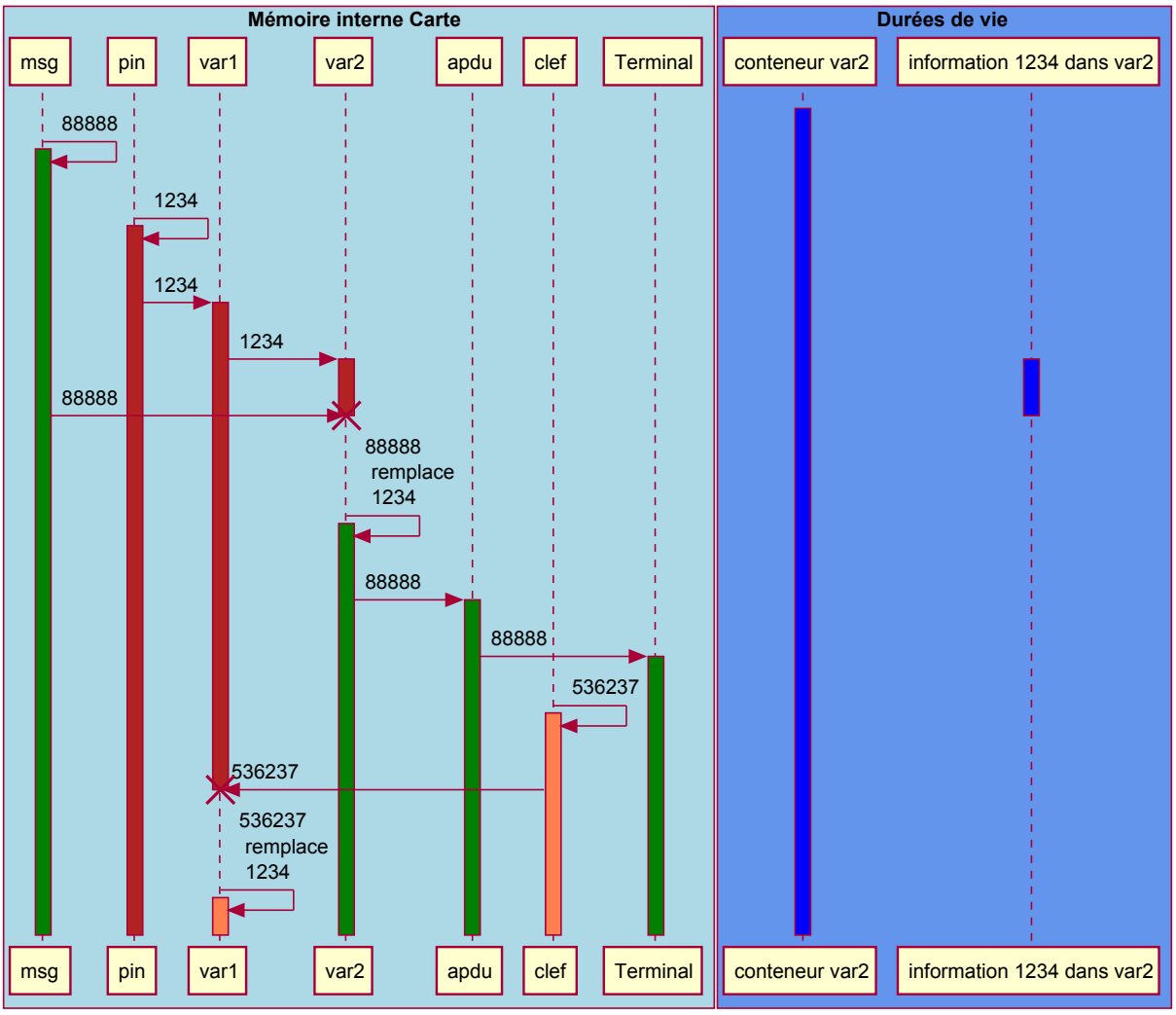


FIGURE 2.5 – Exemple de durée de vie d'un conteneur et durée de vie d'une information à l'intérieur d'un conteneur

On note $C \leftarrow I$ l'ensemble des informations I contenues par C pendant cette durée de vie. Cette notion est liée mais pas strictement équivalente à l'existence d'une écriture pour cette variable. Par exemple, dans le listing 2.7, $var2 \leftarrow I =$ "?","1234","88888" car, avant d'être initialisée à "1234", la variable $var2$ pointe sur une case mémoire qui contient une information inconnue "?". En analysant le reste du programme, il serait possible de déduire la valeur de cette information comme la dernière écriture effectuée dans ce conteneur. L'information "1234" est ensuite remplacée par "88888" lors du transfert d'information à la ligne 8. $C \leftarrow I$ est donc défini par les écritures dans le conteneur qui sont les moyens de modifier les informations contenues à partir du code source.

2.3.1 Confidentialité sous attaque

Nous définissons, au niveau du code source, la propriété de confidentialité suivante.

Propriété 1 (Confidentialité d'une information en cas d'attaque) *Soient C un conteneur parmi l'ensemble des variables $\mathbb{V}$ du programme, $\mathcal{M}$ un modèle d'attaque définissant les attaques possibles, O un conteneur dans l'ensemble des variables observées $\mathbb{V}_O$, on dit que les informations abritées par C sont confidentielles sous attaque si et seulement si*

$$\forall C \in \mathbb{V},\ \forall O \in \mathbb{V}_O,\ \forall I \text{ information telle que } C \leftarrow I, \text{ on a } I \notin Obs(O, \mathcal{M})$$

On note cette propriété :

$$\boxed{Confidentialité(C, O, \mathcal{M})}$$

Autrement dit, quelle que soit l'information abritée dans un conteneur sensible, cette information ne doit jamais se trouver dans un conteneur observé par l'attaquant, même en cas de perturbation.

Par exemple, il peut être intéressant de garantir la confidentialité de `pin` pour le code d'authentification du listing 2.7. Si l'on considère `apdu` comme un élément potentiellement observable par l'attaquant, on peut exprimer la confidentialité de `pin` sous la forme :

$$Confidentialité(pin, apdu, \mathcal{M})$$

Dans cette notation, C_{pin} a été simplifié à pin et C_{apdu} à $apdu$.

Cependant pour pouvoir isoler avec précision une information à partir d'un conteneur, il est nécessaire de définir une zone dans le code dans laquelle nous considérons que les informations, lues à partir du conteneur, sont retenues pour la propriété de confidentialité. En effet, hors d'une telle zone, un conteneur peut contenir des informations peu importantes du point de vue de la confidentialité.

2.3.2 Zone de génération d'information d'un conteneur

Un conteneur peut abriter des informations différentes pendant une durée de temps. Or cette durée peut être associée à des lignes de code successives créant une zone de code.

Définition 6 (Zone de génération) *Une zone de génération Z pour un conteneur C est une zone de code qui s'exprime en ligne à l'aide d'une ligne de début ld et une ligne de fin lf. On adopte la notation suivante. $Z = ld : lf$ pour définir cette zone de code.*

Listing 2.8– Exemple d'une implémentation d'authentification basique avec une tache fonctionnelle sécuritaire sensible mais sans code contre attaques physiques

```
void Authentification(void)                                                    1
{                                                                              2
    int jeton_auth = 0x0F;                                                     3
    int user_pin, card_pin, pin_cpt;                                           4
    user_pin = get_pin();                                                      5
    card_pin = read_eeprom(&ee_card_pin);          // zone de génération       6
    pin_cpt = read_eeprom(&ee_pin_cpt);                                        7
    if (user_pin == card_pin) && (pin_cpt < 3)                                 8
    {                                                                          9
        return jeton_auth;                                                     10
    }                                                                          11
    else                                                                       12
    {                                                                          13
        pin_cpt++;                                                             14
    }                                                                          15
    write_eeprom(&ee_pin_cpt, pin_cpt);                                        16
}                                                                              17
```

De cette définition découle la définition suivante :

Définition 7 (Ensemble d'information d'un conteneur dans une zone de code) *L'ensemble des informations I contenues dans C dans une zone de code Z est noté $C \xleftarrow{Z} I$.*

Ces zones vont permettre de préciser les informations sensibles à considérer pour un conteneur générique qui peut contenir à la fois des informations sensibles et non sensibles.

Propriété 2 (Confidentialité restreinte en cas d'attaque) *Soient C un conteneur parmi l'ensemble des variables $\mathbb{V}$ du programme, $\mathcal{M}$ un modèle d'attaque, O un conteneur dans l'ensemble des variables observées $\mathbb{V}_O$, Z une zone de code, on dit que les informations abritées par C dans la zone Z sont confidentielles sous attaque si et seulement si*

$$\forall C \in \mathbb{V},\ \forall O \in \mathbb{V}_O,\ \forall I \text{ information telle que } C \xleftarrow{Z} I,\ \text{on a } I \notin Obs(O, \mathcal{M})$$

On note cette propriété :

$$\boxed{Confidentialité(C, O, \mathcal{M}, Z)}$$

Autrement dit, quelle que soit l'information contenue dans un conteneur sensible, pendant une durée correspondant à l'exécution du code dans une zone, cette information ne doit jamais se trouver dans un conteneur observé par l'attaquant même en cas d'attaque.

En appliquant cette propriété sur le listing 2.8, la zone de génération d'information sensible pourrait être la zone entre les lignes 6 et 6. La propriété appliquée à ce code source s'écrit :

$$Confidentialité(card\_pin, jeton\_auth, \mathcal{M}, L6 : L6)$$

Dans ce code, seule la ligne 5 est considérée par le développeur comme pertinente du point de vue de la confidentialité de l'information présente dans la variable `card_pin`. On suppose que `jeton_auth` peut être communiqué à l'extérieur du programme et donc constitue un vecteur de fuite d'information.

2.3.3 Zone de garantie de confidentialité

En plus de la zone de génération d'information, on peut restreindre la zone d'application où l'on considère que la propriété de confidentialité doit rester valide. Cette restriction est nécessaire afin de permettre au développeur de préciser plus finement une zone d'intérêt dans laquelle il serait nécessaire d'assurer la confidentialité d'une information pour la sécurité. Cette notion de zone de garantie a été définie précédemment dans la définition 4 de la section 2.2.3. Cette zone permet au développeur de limiter temporellement les conteneurs considérés observables par l'attaquant. On peut redéfinir la propriété de confidentialité sur le code en incorporant cette notion de zone de garantie.

Propriété 3 (Confidentialité restreinte en cas d'attaque garantie sur une zone) *Soient C un conteneur parmi l'ensemble des variables $\mathbb{V}$ du programme, $\mathcal{M}$ un modèle d'attaque, O un conteneur dans l'ensemble des variables observées $\mathbb{V}_O$, soient Z une zone de génération d'information et Z_G une zone de garantie, on dit que les informations manipulées dans la zone Z par C sont confidentielles dans le champ Z_G dans l'hypothèse d'une attaque physique si et seulement si pour toute information I contenue dans C pour un temps correspondant à l'exécution du code dans Z :*

- *le conteneur O observable par l'attaquant n'appartient pas à la zone de code Z_G même en cas d'attaque physique ;*
- *ou ce conteneur appartient à la zone Z_G et l'information I n'est pas observable dans ce conteneur même si le code est attaqué par une attaque de type $\mathcal{M}$.*

La phrase précédente peut être formalisée de la manière suivante :

$$\forall C \in \mathbb{V},\ \forall O \in \mathbb{V}_O,\ \forall I \text{ information telle que } (C \xleftarrow{Z} I),\ on\ a :$$
$$O \notin Z_G \vee (O \in Z_G \wedge I \notin Obs(O, \mathcal{M}))$$

Cette dernière expression se simplifie en :

$$\neg\ (O \in Z_G \wedge I \notin Obs(O, \mathcal{M}))$$

On note cette propriété :

$$\boxed{Confidentialité(C, O, \mathcal{M}, Z, Z_G)}$$

En appliquant cette propriété sur le listing 2.9, la zone de garantie concernée est la zone entre les lignes 8 et 15. En effet, le développeur souhaite que l'information contenue dans la variable `card_pin` reste confidentielle pendant sa comparaison avec `user_pin` et ne soit pas dévoilée hors de la fonction avec `jeton_auth`. On peut écrire cette propriété :

$$\boxed{Confidentialité(user\_pin, jeton\_auth, \mathcal{M}, L6 : L6, L8 : L15)}$$

Les zones de génération et de garantie doivent être fournies par un développeur. En s'appuyant sur l'implémentation de son code, celui-ci doit déterminer la sensibilité des informations manipulées pour établir les zones de génération. Les zones de garanties sont créées à partir de l'emplacement des vecteurs de fuite d'information dans le code. Ces zones peuvent être disjointes. En effet, elles représentent des notions différentes : Z est une zone de génération d'informations sensibles du point de vue de la confidentialité tandis que Z_G est une zone de garantie souhaitée par le développeur où ces informations confidentielles ne doivent pas être visibles au travers de conteneurs observables par l'attaquant.

Listing 2.9– Exemple d'une implémentation d'authentification basique avec une tache fonctionnelle sécuritaire sensible mais sans code contre attaques physiques

```
void Authentification(void)                                                    1
{                                                                              2
    int jeton_auth = 0x0F;                                                     3
    int user_pin, card_pin, pin_cpt;                                           4
    user_pin = get_pin();                                                      5
    card_pin = read_eeprom(&ee_card_pin);          // zone de génération       6
    pin_cpt = read_eeprom(&ee_pin_cpt);                                        7
    if (user_pin == card_pin) && (pin_cpt < 3)     // début de zone de garantie 8
    {                                                                          9
        return jeton_auth;                                                     10
    }                                                                          11
    else                                                                       12
    {                                                                          13
        pin_cpt++;                                                             14
    }                                              // fin de zone de garantie  15
    write_eeprom(&ee_pin_cpt, pin_cpt);                                        16
}                                                                              17
```

2.3.4 Avantage pour l'attaquant et granularité de l'information

Cette définition de la confidentialité ne prend pas en compte le fait qu'une information puisse être chiffrée. En effet, si une information secrète apparaît dans un conteneur observable par l'attaquant la confidentialité de cette information n'est pas violée si celle-ci est chiffrée. À part dans des cas de cryptographie en boîte blanche [Joye 2008], une information telle qu'une clé de chiffrement ou de déchiffrement est dévoilée au moment de son utilisation. Afin de simplifier notre formalisation, nous n'introduirons pas de granularité supplémentaire sur l'information d'un conteneur. Nous considérons que lors du choix des zones de génération, le développeur sélectionne des zones de code pour un conteneur alors que celui-ci contient une information déchiffrée. Cependant, si on souhaite prendre en compte la granularité de l'information dans la propriété de confidentialité, on peut redéfinir la fonction d'observation comme suit :

Définition 8 (Redéfinition de la fonction d'observation) *Soient O un conteneur observé parmi l'ensemble des variables, $\mathcal{M}$ un modèle d'attaque, $\mathfrak{F}()$ une fonction transformant une information I telle que $\forall(C \leftarrow I)$, on a :*

$$\boxed{I \notin Obs(O,\mathcal{M}) \iff \mathfrak{F}(I) \cap Obs(O,\mathcal{M}) = \varnothing}$$

Le listing 2.10 donne un exemple de déchiffrement d'une information enregistrée chiffrée en EEPROM pour son utilisation dans une comparaison. Si l'attaquant est capable d'observer le conteneur `pin_carte` à la ligne 5, il a accès à l'information secrète. Cependant, à ce moment de l'exécution, cette information est chiffrée donc sa confidentialité n'est pas compromise. D'un autre côté, si l'attaquant est capable d'observer le conteneur `pin_carte` à la ligne 9, il a accès à l'information secrète en clair et donc sa confidentialité est compromise.

Listing 2.10– déchiffrement d'une information confidentielle

```
void verif_pin()
{
    int jeton_auth = 0x0F;
    int pin_carte;
    pin_carte = lecture_eeprom(&pin_carte_chiffre);
    int pin_utilisateur;
    pin_utilisateur = lecture_terminal();
    // déchiffrement de pin_carte_chiffre
    pin_carte = dechiffre(pin_carte_chiffre);
    if (pin_utilisateur == pin_carte)
    {
        return jeton_auth;
    }
}
```

A titre d'information, dans le cas spécifique d'une comparaison, on peut éviter de déchiffrer une information sensible en chiffrant l'élément comparé comme dans le listing 2.11. Ainsi, l'information secrète n'est jamais déchiffrée et un attaquant en observant le conteneur `pin_carte` à n'importe quel moment de l'exécution n'a jamais accès à l'information en clair. Cette technique permet de sécuriser l'information secrète et d'assurer sa confidentialité.

Listing 2.11– Comparaison chiffrée d'une information confidentielle

```
void verif_pin()
{
    int jeton_auth = 0x0F;
    int pin_carte;
    pin_carte = lecture_eeprom(&pin_carte_chiffre);
    int pin_utilisateur;
    pin_utilisateur = lecture_terminal();
    // chiffrement de pin_utilisateur
    pin_utilisateur = chiffre(pin_utilisateur);
    if (pin_utilisateur == pin_carte)
    {
        return jeton_auth;
    }
}
```

En rajoutant une telle granularité à l'information, on peut décrire de manière précise les cas qui avantagent l'attaquant. En effet, sans cette granularité, on peut seulement dire que l'attaquant a pris connaissance d'une information sensible. Or, si celle-ci est chiffrée, l'attaquant n'en tire aucun avantage.

On peut faire l'hypothèse qu'on se place volontairement dans des cas où les informations sont déchiffrées, à la suite ou non d'une attaque, pour l'observation des conteneurs et la vérification de la propriété de confidentialité. On peut aussi prendre en compte le fait qu'une information peut être chiffrée ou non lors de la formalisation grâce à cette granularité de l'information. On peut ainsi exprimer le gain pour l'attaquant :

$$\boxed{\neg\, Confidentialité(C, O, \mathcal{M}, Z, Z_G) \wedge I_{clair} \in Obs(O, \mathcal{M})}$$

avec I_{clair}, une information secrète en clair c'est-à-dire non chiffrée.

Ainsi, au niveau du code, on doit spécifier des APIs de transformation pouvant être formalisées de la manière suivante :

$$C^0 < I_{chiffre} > \curvearrowleft f(C^1 < I_{clair} >)$$

et

$$C^2 < I_{clair} > \curvearrowleft f^{-1}(C^3 < I_{chiffre} >)$$

avec $C < I >$ un conteneur C abritant I à un instant donné, $B \curvearrowleft A$ un transfert d'information du conteneur A vers le conteneur B, $f()$ une fonction de transformation, $f^{-1}()$ son inverse. Par exemple dans le listing 2.11 à la ligne 9, $f()$ = `chiffre()` et dans le listing 2.10 à la ligne 9, $f^{-1}()$ = `dechiffre()`.

Ces fonctions de transformations permettent notamment d'isoler dans le code les endroits où une information secrète est déchiffrée et de se concentrer sur ces portions de code pour l'analyse et la vérification de la confidentialité de cette information.

2.4 Intégrité

Dans cette section, nous nous intéressons à l'intégrité d'exécution d'un programme sous attaque. Par rapport à la propriété précédente, nous ne poussons pas aussi loin le formalisme, le but étant de clarifier les notions d'intégrité.

2.4.1 Accès à un conteneur et zone de garantie

Pour accéder à l'information abritée dans un conteneur, on a recours à une opération de lecture qui copie l'information du conteneur dans un autre conteneur pour pouvoir l'utiliser. Pour remplacer l'information abritée dans un conteneur, on a recours à une opération d'écriture qui écrasera l'information abritée par une nouvelle information.

Définition 9 (Accès en lecture et écriture à un conteneur) *On note respectivement $r(C)$ et $w(C)$, les opérations de lecture et d'écriture d'un conteneur C ; $\overline{r(C)}$ et $\overline{w(C)}$ sont respectivement les valeurs lues et écrites associées à chacune d'elles.*

A partir de ces définitions, on peut dire que l'exécution d'un code se résume à des séquences d'opérations de lecture et d'écriture de multiples conteneurs, d'opérations logiques et arithmétiques et des branchements de flot. En cas d'attaque physique, comme présenté en section 2.2.4, l'attaquant peut modifier l'information stockée dans un conteneur, autrement dit effectuer une opération d'écriture sur ce conteneur ou sauter des instructions consécutives du code. Le chapitre 3 développera ces éléments. Nous supposons également que l'attaquant n'a le droit qu'à un unique essai pour son attaque, ce qui est raisonnablement le cas le plus répandu à l'heure actuelle. En cas d'attaque contre un conteneur, le programme modifie la séquence des lectures/écritures concernant celui-ci. Dans cette nouvelle séquence, une opération additionnelle d'écriture de ce conteneur a été insérée ou une ou plusieurs opérations de lecture / écriture auront été enlevées. On note $w_a(C)$ cette écriture additionnelle et $S(C) = [x_1, x_2, \cdots x_n]$ avec $x_i = r$ ou $x_i = w$ une séquence ordonnée de lectures / écritures de C.

Si une opération d'écriture additionnelle est insérée, 4 scénarii sont possibles en fonction de l'emplacement de l'attaque dans la séquence. Ils peuvent être regroupés en deux comportements :

- $S_{1a}(C) = [r, w_a, r]$ *et* $S_{1b}(C) = [w, w_a, r]$: l'écriture additionnelle modifie la valeur lue par la seconde lecture ;

– $S_{2a}(C) = [r, w_a, w]$ *et* $S_{2b}(C) = [w, w_a, w]$: l'écriture additionnelle n'a pas d'effet car l'écriture suivante annule son effet.

Seuls les cas $S_{1a}(C)$ et $S_{1b}(C)$, où l'écriture additionnelle provoque une modification de l'intégrité de la valeur lue, nous intéressent. En supposant que le code fonctionnel du projet est correct et donc qu'il n'existe pas de variable lue avant d'être initialisée, on peut dire que dans une séquence $S(C)$, la lecture d'un conteneur est toujours précédée par une écriture de celui-ci.

On peut donc déduire que, les seules zones intéressantes pour vérifier une propriété d'intégrité d'un conteneur à l'exécution sous attaque, sont les zones de type ($S_{1a}(C)$ et $S_{1b}(C)$). On peut englober ces deux types de zones dans une zone générique débutant par une écriture de ce conteneur suivie d'une ou plusieurs lectures de ce même conteneur. La dernière lecture x_0 doit correspondre à une lecture fonctionnelle dont on souhaite garantir l'intégrité. On note Z_G cette zone de code de garantie. Cette zone est formalisée par :

$$S_{Z_G}(C) = [w_0, x_1, x_2, \cdots x_n, x_0] \; avec \; x_i = r$$

Illustrons la définition de cette zone de garantie à partir du code fonctionnel du listing 2.12. Ce listing réalise la lecture EEPROM du conteneur `card_pin` dans le but de l'utiliser pour une comparaison. On souhaite, dans cet exemple, garantir l'intégrité de ce conteneur afin d'être sur que l'information utilisée dans la comparaison n'est pas faussée.

Listing 2.12– Zone de garantie souhaitée pour l'intégrité du conteneur card_pin sous attaque

```
void Authentification(void)                                          1
{                                                                    2
    int jeton_auth = 0x0F;                                           3
    int user_pin, card_pin, pin_cpt;                                 4
    user_pin = get_pin();                                            5
    card_pin = read_eeprom(&ee_card_pin);       // début de zone de garantie  6
    pin_cpt = read_eeprom(&ee_pin_cpt);                              7
    if (user_pin == card_pin) && (pin_cpt < 3)  // fin de zone de garantie  8
    {                                                                9
        return jeton_auth;                                           10
    }                                                                11
    else                                                             12
    {                                                                13
        pin_cpt++;                                                   14
    }                                                                15
    write_eeprom(&ee_pin_cpt, pin_cpt);                              16
}                                                                    17
```

Si, dans le listing 2.12, on souhaite garantir l'intégrité du conteneur `card_pin` à partir de sa première affectation mémoire jusqu'à sa comparaison avec le PIN présenté par l'utilisateur, une zone de garantie spécifique doit être définie. Le contexte sensible du point de vue de la sécurité débute au moment de la première écriture d'une information sensible dans ce conteneur. Ce contexte débute donc en ligne 6, où la valeur est lue depuis l'EEPROM. Ce contexte se termine à la ligne 8, où sa valeur est comparée et permet en cas de succès d'obtenir un jeton d'authentification. Ainsi, dans ce cas, on a : $Z_G = L6 : L8$.

2.4.2 Intégrité en lecture et écriture

Pour établir la propriété d'intégrité d'un conteneur sous attaque physique, nous devons nous assurer que les opérations de lecture ou écriture d'un conteneur dans la zone considérée s'effectuent correctement. Sans attaque, le programme s'effectue correctement par hypothèse sur

le compilateur et sur la plate-forme d'exécution. Dans le cas d'une attaque, la perturbation peut agir sur une lecture ou une écriture ou un calcul intermédiaire conduisant à une mauvaise écriture. Notre définition d'intégrité de conteneur cherche à exprimer le fait que le programme s'exécute correctement vis à vis du code manipulant ce conteneur. La réponse sécuritaire peut aussi être intégrée à cette définition comme nous l'avons fait pour la propriété de confidentialité. Dans un premier temps, on définit cette notion sans considérer les contre-mesures.

Le but est de définir l'intégrité sur une séquence d'instructions manipulant un conteneur C. Pour cela, on décompose une séquence en opérations atomiques de lectures/écritures sur ce conteneur. Une attaque ne viole pas l'intégrité d'une opération de lecture si la valeur lue est identique à celle qui serait lue sans attaque. On définit de même l'intégrité d'une opération d'écriture.

On note $X \not\rightarrow^a$ l'intégrité d'une opération X sous attaque a.

Définition 10 (Intégrité de lecture d'un conteneur) *Pour une attaque donnée a, on dit qu'une lecture du conteneur C est intègre vis à vis de a si et seulement si la valeur lue sous attaque est la même que celle lue sans attaque.*

$$r(C) \not\rightarrow^a \iff \overline{r_a(C)} = \overline{r(C)}$$

où $r_a(C)$ représente la lecture de C dans le programme attaqué par a.

Définition 11 (Intégrité d'écriture d'un conteneur) *De même pour l'écriture, on définit l'intégrité de $w(C)$ par :*

$$w(C) \not\rightarrow^a \iff \overline{w_a(C)} = \overline{w(C)}$$

où $w_a(C)$ représente l'écriture dans C dans le programme attaqué par a.

On souhaite maintenant définir l'intégrité d'un conteneur, sous attaque physique, en prenant en compte un réponse sécuritaire KC du système pouvant survenir plus tard dans l'exécution du programme. On note l'appel à cette réponse suivant la convention LTL de logique temporelle $F(KC)$ c'est-à-dire que KC doit survenir plus tard dans l'exécution du programme.

On peut maintenant établir la propriété suivante pour l'intégrité :

Propriété 4 (Intégrité d'un conteneur à l'exécution sous attaque) *Soient a une attaque parmi l'ensemble $\mathbb{A}$ des attaques considérées, C un conteneur parmi l'ensemble $\mathbb{V}$ des variables, Z_G une zone de garantie définie par une écriture suivie d'une ou plusieurs lectures de C : $S_{Z_G}(C) = [w_0, r_1, \ldots r_k]$ et KC la réponse sécuritaire, l'intégrité de C est garantie dans Z_G sous attaque si et seulement si*

$$\forall a \in \mathbb{A}, \text{ on } a : \left\{ \begin{array}{r} w_a \notin S_{Z_G}(C) \\ w_0 \not\rightarrow^a \\ \forall i \in [1, \ldots, k] \quad r_i \not\rightarrow^a \end{array} \right\} \vee F(KC)$$

On note cette propriété :

$$\boxed{Intégrité(C, Z_G, a)}$$

Pour un ensemble d'attaques A, on définit naturellement l'intégrité d'un conteneur vis à vis de cette famille par :

$$\mathit{Intégrité}(C, Z_G, \mathbb{A}) \iff \forall a \in \mathbb{A},\ \mathit{Intégrité}(C, Z_G, a)$$

Cette propriété d'intégrité peut s'interpréter de la manière suivante : soit une zone de code débutant par l'affectation d'une variable à une valeur (écriture) dont on souhaite garantir l'intégrité pour une utilisation future (lecture). On souhaite avoir cette garantie en présence d'une attaque physique. Si la conséquence de cette attaque physique impacte la variable considérée, cette conséquence est équivalente à : l'insertion d'une opération d'écriture sur cette variable à un endroit du code ou le saut d'une à plusieurs instructions affectant cette variable. Nous justifions cette affirmation dans notre modèle d'attaque que nous développons dans le chapitre 3. Garantir l'intégrité de la variable à partir de son affectation jusqu'à son utilisation revient donc à s'assurer que :

- la première affectation n'a pas été sautée sinon lors de l'utilisation la variable a pour valeur la dernière valeur abritée par le conteneur lu ;
- la première affectation reçoit la bonne valeur ;
- chaque lecture dans la zone n'est pas sautée sinon l'intégrité de lecture, lors de l'utilisation fonctionnelle, est violée ;
- chaque lecture lit la valeur attendue ;
- l'attaque n'introduit pas l'équivalent d'une écriture de ce conteneur dans la zone considérée. En effet, si cette écriture induite par l'attaque apparaît *avant* la zone, elle est écrasée par la première écriture (considérée comme correcte) de la zone. Si celle-ci apparaît *après* la fin de la zone, elle n'affecte pas la dernière lecture du conteneur dans la zone et par conséquent son utilisation, ce qui est ce que nous cherchons à garantir.

Si une des conditions précédentes n'est pas respectée, le système doit réagir de manière appropriée et déclencher une réponse sécuritaire. Si une telle réponse est déclenchée, on estime que l'intégrité du conteneur considéré est assurée.

Généralisation à une exécution L'intégrité telle que définie par la définition 4 ne prend en compte qu'un seul conteneur dans une zone précise Z_G délimitée par un écriture et une lecture de ce conteneur. Or une séquence d'exécution est composée de plusieurs séquences délimitées par de multiples accès à ce conteneur. Garantir l'intégrité d'un même conteneur tout au long d'une exécution revient à garantir l'intégrité de celui-ci sur l'ensemble des zones de garantie Z_G.

Généralisation à de multiples conteneurs et dépendance : Application à l'intégrité d'exécution Pendant un flot d'exécution, un conteneur peut dépendre de plusieurs autres conteneurs en termes de dépendance d'information. Dans l'instruction `x = a + b;`, on voit qu'à l'exécution de cette ligne de code, l'information contenue dans la variable `x` à la suite de `a` et `b`. Si on considère le code depuis le début de l'exécution du programme jusqu'à l'interprétation de cette instruction, on peut établir que l'intégrité de cette instruction dépend de l'intégrité de l'ensemble des conteneurs qui influent sur cette instruction par dépendance d'information ainsi que l'ensemble du code logique (conditions, boucles, sauts, appels) menant à cette instruction.

Si on fait l'hypothèse que le code logique est valide et n'est pas influencé par une attaque physique, on peut dire que l'intégrité de l'exécution du code depuis le début de l'exécution

jusqu'à une instruction est garantie si l'ensemble des conteneurs impliqués dans cette exécution sont intègres.

2.4.3 Avantage pour l'attaquant

L'avantage que peut tirer un attaquant d'une modification en intégrité d'un conteneur (et par conséquent de l'information qu'il abrite) dépend des spécifications fonctionnelles du produit et donc du sens associé à l'information. En effet, supposons deux variables servant de compteurs `cpt1` et `cpt2` : `cpt1` comptabilise le nombre d'essais effectués par l'utilisateur pour s'authentifier en présentant un PIN, `cpt2` comptabilise le plafond maximum de retrait de la carte. Un attaquant a un intérêt à minimiser `cpt1` mais à maximiser `cpt2`. Ainsi compromettre l'intégrité de ces variables n'est avantageux pour l'attaquant uniquement si cela lui permet d'aller à l'encontre d'un comportement fonctionnel lié à la sécurité.

2.4.4 Intégrité d'exécution

Le but de l'intégrité d'exécution, pour le développeur, est de garantir qu'au niveau fonctionnel certaines opérations menant à la réalisation d'une tâche sont bien effectuées. Le premier problème pour le développeur est l'identification de ces opérations critiques d'un point de vue fonctionnel et d'un point de vue de la sécurité. Prenons pour exemple la vérification du PIN dans laquelle une partie du code enregistre des rapports pour auditer le code en cas d'erreurs. Ce code n'est pas critique vis-à-vis de la vérification, cependant incrémenter le compteur d'essai du PIN l'est. Le développeur est seulement intéressé par la garantie des instructions du code contribuant de manière effective à la réalisation d'une fonctionnalité principale. Dans cet exemple, il s'agit de la vérification ou de sa sécurité. Vérifier la sécurité de mécanismes supplémentaires, qui n'ont pas directement trait à la fonctionnalité ou sa sécurité, n'est pas utile pour le développeur.

Dans la section précédente, nous avons défini l'intégrité d'un conteneur sous attaque physique. Dans cette section, nous allons élargir ce concept d'intégrité à une instruction de code. Une instruction de code est n'importe quelle instruction du langage de programmation, ici le C. Pour réaliser une tâche, certaines instructions sont critiques.

Dans la section 2.4.2, on a défini $r(C) \not\mapsto$ et $w(C) \not\mapsto$, l'intégrité d'une opération de lecture ou d'écriture d'un conteneur C. Nous pouvons étendre cette définition à n'importe quelle instruction du langage de programmation en prenant en compte le flot d'exécution et en introduisant le concept de saut. En effet, les opérations de lecture et d'écriture telles que nous les avons définies jusqu'à présent ne supposent pas l'existence d'un flot d'instructions. Cependant, on peut définir un flot d'instructions comme une suite de transformations d'informations et de conteneurs suivie d'un saut vers une autre instruction $Instr'$ à exécuter noté $\curvearrowright Instr'$. On peut formaliser cette nouvelle propriété sur l'intégrité d'une instruction $Instr$ de la manière suivante. Cette formalisation se base sur les définitions d'un état mémoire du système et de la transition d'états mémoires.

Définition 12 (État mémoire) *L'état mémoire E d'un système à un instant t est défini par l'ensemble des couples (C, I) avec C un conteneur et I l'information abritée dans ce conteneur à l'instant t*

Définition 13 (Transition d'état mémoire) *L'exécution d'une instruction permet de passer d'un état mémoire à un autre avec pour modification les conséquences de l'instruction sur la*

mémoire. On note $E \underset{Instr}{\multimap} E'$ *la transition entre un état* E *et un état* E' *suite à l'exécution d'une instruction* $Instr$.

On peut alors établir la propriété suivante :

Propriété 5 (Intégrité d'une instruction) *Soient* E_1 *l'état mémoire du système avant l'exécution de* $Instr$, E_2 *l'état mémoire du système après l'exécution de* $Instr$. *Soit* $Instr'$ *l'instruction qui suit* $Instr$ *dans le flot d'exécution du code. On a l'intégrité d'exécution de l'instruction* $Instr$ *si et seulement si les deux conditions du système suivant sont remplies.*

$$Instr \leadsto\!\!\!\!\!\!\Longleftrightarrow \begin{cases} E_1 \underset{Instr}{\multimap} E_2 \\ \curvearrowright Instr' \end{cases}$$

Listing 2.13– Intégrité d'exécution d'une condition en C

```
instr_avant();
if (x == 0)
{
    then_exec();
} else
{
    else_exec();
}
instr_apres();
```

Ainsi pour une condition en C, i.e., un `if` comme implémenté dans le listing 2.13, l'intégrité de cette instruction de condition peut s'appliquer sur le code de la manière suivante :

$$If \leadsto\!\!\!\!\!\!\Longleftrightarrow \begin{cases} E_{L2} \underset{If}{\multimap} E_{L8} \\ \curvearrowright Instr : L9 \end{cases}$$

Cette définition peut être aussi être étendue aux boucles. Par exemple, prenons la boucle implémentée en 2.14.

Listing 2.14– Intégrité d'exécution d'une boucle en C

```
int i;
int y = 1;
instr_avant();
for (i=0; i<10; i++)
{
    y++;
}
instr_apres();
```

On peut appliquer la propriété d'intégrité d'exécution à cette boucle.

$$For \leadsto\!\!\!\!\!\!\Longleftrightarrow \begin{cases} E_{L4} \underset{For}{\multimap} E_{L8} \\ \curvearrowright Instr : L8 \end{cases}$$

Un appel de fonction, dont on peut trouver une implémentation dans le listing 2.15, est aussi pris en compte par la propriété.

Listing 2.15– Intégrité d'exécution d'un appel de fonction en C

```
int x = 5;                    1
int y;                        2
instr_avant();                3
y = f(x);                     4
instr_apres();                5
```

Listing 2.16– Intégrité d'exécution d'un appel de fonction en C

```
void f(int x)                 10
{                             11
    x++;                      12
    return x;                 13
}                             14
```

Pour cet appel de fonction, on écrirait :

$$Call \rightsquigarrow \Longleftrightarrow \begin{cases} E_{L4} \underset{Call}{\multimap} E_{L5} \\ \curvearrowright Instr : L5 \end{cases}$$

Ces propriétés, déclinées sur le code source, permettent d'exprimer des problématiques de sécurité uniquement à partir d'une représentation mémoire des informations. Il est maintenant possible en vérifiant des états mémoires du système de vérifier des garanties de sécurité sur le code source. Grâce à de telles propriétés, un développeur est capable d'exprimer les besoins de sécurité de son application en terme de confidentialité et d'intégrité.

2.5 Conclusion

Nous avons formalisé le problème de sécurité à travers différentes propriétés faisant intervenir des éléments de code et dans certains cas la réponse sécuritaire du système. L'adoption de cette perspective permet au développeur d'exprimer ses besoins en terme de garanties de sécurité sur son code. Disposer de ce formalisme permet aussi de cerner l'ensemble de la sécurité fonctionnelle du système en restant au niveau du code source de haut niveau. Le fait de garder cette perspective présente un réel avantage pour le développeur de façon à abstraire la complexité d'un code de bas niveau. Un avantage sous-jacent de la représentation choisie est que la notion de conteneur et d'information peut être projeté aussi bien au niveau matériel qu'au niveau logiciel. Cette cohésion permet d'unifier une représentation assembleur et C. Elle permet également aux propriétés d'être appliquées de manière transparente par rapport au langage utilisé car il est toujours possible pour une information donnée de trouver le conteneur correspondant. Les propriétés de sécurité ont aussi été créées de manière à regrouper les possibilités d'attaque au sein même de la propriété. Cette particularité fait que ces propriétés se prêtent bien à la vérification de la sécurité sous attaques physiques. De plus, ces propriétés ont été pensées de manière à faciliter l'implémentation des vérifications concrètes en testant la correspondance des états mémoires obtenus avec et sans attaque. Enfin, ces propriétés de sécurité sont indépendantes de la technique d'analyse et de vérification utilisées ce qui les rend particulièrement adaptées pour exprimer génériquement des besoins de sécurité au niveau du code source. Cette formalisation devrait pouvoir s'appliquer à n'importe quelle architecture embarquée voire à n'importe quel code soumis à des attaques par fautes physiques.

Des limitations sont cependant associées aux propriétés de sécurité définies dans ce chapitre. En effet, ces propriétés traitent séparément la confidentialité et l'intégrité des informations. Cette séparation permet l'utilisation d'un modèle proche de la mémoire lors de leur définition. Elle empêche aussi toute vérification simultanée des deux propriétés potentiellement liées par

une relation de causalité, ce qui peut être souhaité lors de la résolution de preuves. Il est aussi important de noter que ces définitions imposent plusieurs limitations :

- La confidentialité de l'exécution du code n'est pas considéré du point de vue fonctionnel ;
- La sémantique du code et ses conséquences sur la confidentialité des informations ne sont pas prises en compte ;
- Le modèle d'attaquant est limité de manière réaliste dans ses capacités d'observation et d'action.

Pour des formalisations qui lèvent les deux dernières limitations, le lecteur peut se référer à [Myers *et al.* 2004] et [Balliu & Mastroeni 2009]. Les auteurs proposent des notions de *déclassification* d'information et de *robustesse* aux attaques particulièrement intéressantes dans un contexte cryptographique. L'approche adoptée ici est plus simple mais a priori plus facile à mettre en pratique.

En effet, les limitations sont associées à plusieurs avantages qui rendent l'utilisation des propriétés plus faciles à exploiter. L'intégrité d'une information repose sur l'intégrité du conteneur qui l'abrite. Cette définition permet de seulement considérer des séquences de lectures et d'écritures de variables dans le code. Cette définition est associée à celle d'intégrité d'exécution d'une instruction ou suite d'instructions. Étant donné que cette intégrité d'exécution est liée à des états mémoires du système, il est toujours possible de créer des images totales ou partielles de la mémoire à des instants de l'exécution. Cette particularité simplifie l'élaboration de vérifications pour cette propriété et autorise le choix des points de vérification si les informations sont liées entre elles. De plus, en faisant l'hypothèse que la confidentialité d'une information repose sur l'intégrité d'exécution d'appels à des interfaces de programmation (de chiffrement par exemple), l'ensemble des vérifications peut être ramené à l'intégrité d'exécution de code. Des hypothèses similaires permettent de déléguer des vérifications ce qui rend chacune d'elle moins complexe. Le fait de faire de telles hypothèses a aussi l'avantage de permettre de cloisonner les vérifications. Ces hypothèses doivent cependant être faites avec soin et parfois être accompagnées de vérifications de sécurité supplémentaires. Par exemple, les interfaces cryptographiques ne doivent pas utiliser le même tampon d'entrée et de sortie. Le cloisonnement des vérifications et la séparation des propriétés de confidentialité et d'intégrité du point de vue fonctionnel force la création des vérifications de sécurité unitaires qui lors d'une campagne de vérification permet d'isoler et de remonter jusqu'à l'origine d'une vulnérabilité.

Malgré ce travail de formalisation sur les propriétés de sécurité qui sont les priorités d'Oberthur Technologies, ces propriétés ne seront pas réutilisées telles quelles dans les chapitres suivants. En effet, cette thèse a mené le travail de formalisation en parallèle des méthodes proposées ci-après. Les notions de sécurité (confidentialité, intégrité) sur lesquelles sont basées les propriétés établies sont aussi présentes dans les Critères Communs mais ne prennent pas en compte l'implémentation par le développeur. Elles sont exprimées à un niveau fonctionnel, ce qui empêche une utilisation directe sur le code source par celles-ci. Cependant, tout comme les Critères Communs regroupent dans un schéma de sécurité un ensemble d'exigences de sécurité, les propriétés de sécurité établies dans ce chapitre doivent être regroupées pour former une politique de sécurité complète et cohérente sur l'ensemble du code source. De plus, les notations mathématiques doivent encore être adaptées avant une utilisation dans un système informatique afin d'être manipulées par des programmes.

Par ailleurs, une fois cette politique de sécurité écrite, les méthodes développées dans les chapitres qui suivent peuvent se servir de cette politique en entrée des vérifications ou des

tests. Par exemple, pour le chapitre 4, ces propriétés projetées dans un langage de type *ACSL* permettent une vérification statique de celles-ci. Comme nous le verrons plus loin, ce problème est en soi délicat. Par ailleurs, pour le chapitre 5, le test de la propriété est assurée par un oracle (interne ou externe au programme testé) afin de savoir si la propriété est violée. Dans ce cas, il faut explicitement programmer la vérification de la propriété en projetant ses composantes sur le code source, par exemple une variable à tester.

Le chapitre qui suit, s'intéresse maintenant aux attaques et à leurs conséquences bas niveau. Il définit clairement le modèle d'attaque et permet de cerner la nature des attaques considérées dans cette thèse.

2.6 Perspectives

2.6.1 Aspects sémantiques

Les propriétés de sécurité exprimées dans ce chapitre se basent sur une représentation simple proche de la mémoire. Ce point de vue est adopté afin de faciliter sa projection sur le code source tout en mettant les propriétés ainsi exprimées à la portée du développeur. Ce point de vue ignore la sémantique associée au langage de programmation pour se concentrer sur des éléments qui peuvent être testés par une analyse de la mémoire. Rajouter des aspects de sémantiques, notamment les prédicats développés dans [Balliu & Mastroeni 2009] et [Myers *et al.* 2004] peuvent apporter une nouvelle dimension aux propriétés et à leur vérification. La confidentialité se prête bien à cette approche où l'analyse sémantique du code peut permettre d'intégrer des caractéristiques de fuites liées à la sémantique du code source. Ces caractéristiques permettent de déduire, à partir de ce qu'un attaquant apprend par canaux cachés, s'il est capable ou non de compromettre la confidentialité d'informations sensibles.

2.6.2 Temporalité des propriétés

Les propriétés de sécurité établies dans ce chapitre peuvent servir à exprimer, sur du code C, des besoins en sécurité tels que la confidentialité et l'intégrité. Ces propriétés prennent en compte la possibilité d'attaques parvenant à modifier le code source. Si ces propriétés couvrent une partie des besoins en sécurité d'un système, elles ne traitent pas en détail du séquencement des opérations à respecter fonctionnellement. Le séquencement des opérations dans le code peut avoir une importance vis-à-vis des spécifications mais aussi de la sécurité. Exprimer de telles propriétés de logique temporelle et les vérifier sur du code C sous attaque a déjà été traité [Andouard 2009] à l'aide de Model Checking. L'auteur de [Leroy 2004] donne un aperçu du fonctionnement du Model Checking utilisé pour vérifier des propriétés de contrôle de flot. Cependant, une représentation à haut niveau de ces propriétés temporelles ne couvre pas une éventuelle optimisation du compilateur sur l'implémentation du code. Intégrer des aspects temporels aux propriétés de sécurité peut permettre de les étendre et augmenter le pouvoir d'expression du développeur pour exprimer ses besoins en sécurité.

2.6.3 Prise en compte du compilateur

En embarqué, pour des raisons de tailles et de performances, les codes sont fortement optimisés au moment de la compilation. Alors que des codes compilés avec, par exemple, GCC sont par

défaut optimisés à un niveau 0 (`-O0`), les codes embarqués peuvent être optimisés à des niveaux bien supérieurs (`-O9` ou `-O10`, par exemple). Si cette compilation apporte des gains, elle peut parfois être faite au détriment de la sécurité. En effet, les passes d'optimisation suivent des schémas de réarrangement et regroupement de code. Différents exemples de ces passes sont décrites dans la référence [Keil 2012]. Prenons par exemple le schéma de regroupement de code commun en sous blocs pour un gain de taille. Cette optimisation peut, en modifiant l'ordre des opérations, aller à l'encontre de l'intention du développeur ; celui-ci souhaitant une séquence précise d'opération sur laquelle repose la sécurité de son implémentation. L'agencement des registres est aussi faite par le compilateur de manière transparente au niveau du code source et pour le développeur. Le manque de contrôle fin au niveau C de ces aspects d'optimisation peut introduire des failles de sécurité dans le système. Cependant, l'optimisation la plus préjudiciable est sans doute le schéma qui retire le code inutile et notamment les redondances dans le code. En effet, doubler des opérations est une technique très utilisée en tant que contre-mesure. Le compilateur incapable de différencier une redondance souhaitée pour la sécurité d'un code redondant inutile optimise les deux cas de la même manière. Cette dernière optimisation peut très bien retirer une défense implémentée par le développeur. De plus, les passes d'optimisation sont empilées, ce qui peut rendre le sens du code généré très éloigné de celui de départ et difficile à relire. Ainsi, pour vérifier si une sécurité implémentée au niveau C est bien efficace une fois optimisée par le compilateur, le développeur doit s'intéresser au code assembleur généré et vérifier si celui-ci respecte bien les garanties de sécurité souhaitées. Une solution possible à ces problématiques de sécurité est d'optimiser partiellement le code à des niveaux d'optimisation inférieur afin de préserver la sémantique de départ. Les compilateurs peuvent fournir des directives (`#pragma GCC optimize`) permettant cette optimisation partielle. Une autre méthode consiste à modifier le code de départ en introduisant une forme de code que le compilateur ne sait pas optimiser.

Une perspective possible serait d'introduire un aspect temporel aux propriétés de sécurité déjà établies et exprimées sur le code source. En projetant cet aspect sur le code assembleur généré par le compilateur, on pourrait s'assurer que des contraintes de séquencement sont bien respectées même après compilation.

2.6.4 Théorie de l'information

Une dernière perspective pourrait être de raffiner ces propriétés, notamment la propriété de confidentialité, afin de prendre en compte des éléments de la théorie de l'information. On pourrait alors introduire et définir précisément des transferts d'information partiels entre conteneurs ou des transformations d'informations. L'information visible par l'attaquant pourrait ainsi être caractérisée au travers d'un modèle de fuite [Prouff & Rivain 2009]. Ces nouveaux éléments pourraient alors être utilisés pour répondre de manière fine à des problématiques de confidentialité plus complexes et établir des défenses plus sophistiquées [Coron *et al.* 2007].

2.6.5 Génération automatique de propriétés

Une des difficultés à l'adoption généralisée dans un contexte industriel de vérifications de la sécurité par preuves formelles est le temps associé à la création des preuves. Cette difficulté s'applique aussi aux propriétés de sécurité. Une perspective intéressante serait de réussir à générer ces preuves ou propriétés directement depuis les spécifications fonctionnelles. Certains essais

dans ce sens ont déjà été réalisé [Moebius *et al.* 2009]. Une extension de cette perspective est la création d'un compilateur capable à partir de ces preuves de générer le code source associé respectant les contraintes de performance propre à l'embarqué. Le respect de propriétés de sécurité après la compilation peut aussi être effectué à l'aide d'un compilateur certifié [Blazy *et al.* 2006].

Chapitre 3

Modèle d'attaque

Sommaire

3.1 Introduction 81
3.2 Caractérisation de l'attaquant 82
3.2.1 Capacité d'observation 82
3.2.2 Capacité d'action 85
3.3 Modélisation des conséquences à haut niveau 87
3.4 Modèle d'attaque physique à haut niveau 88
3.4.1 Effets des attaques physiques 88
3.4.2 Importance des attaques par NOP et par saut 90
3.4.3 Attaques de code 91
3.4.4 Attaques de données 97
3.4.5 Décalage d'interprétation du code 97
3.5 Conclusion 104
3.6 Perspectives 105
3.6.1 Prise en compte d'attaques d'un ordre supérieur 105
3.6.2 Extension du modèle 107

3.1 Introduction

Établir un modèle d'attaque permet de cadrer les capacités de l'attaquant considéré dans la vérification des propriétés de sécurité. Les propriétés de sécurité envisagées sont décrites dans le chapitre 2. Dans ce chapitre, nous caractérisons un attaquant par sa capacité à observer et à agir sur un système. Dans ce chapitre, nous définissons dans quelle mesure un attaquant peut utiliser ces capacités. Nous adoptons un point de vue particulier consistant à ramener les effets matériels à un niveau d'abstraction supérieur. Nous présentons d'abord d'autres modèles existants et les comparons au modèle proposé. Ensuite, nous exposons notre modèle, ses caractéristiques et ses implications.

Créer un modèle formel, précis et complet des capacités d'un attaquant est nécessaire afin d'apporter une validation correcte de la sécurité. En effet, on peut prendre l'exemple d'une implémentation cryptographique qui, alors qu'elle était prouvée sécurisée, n'a pas résister à la découverte de fautes physiques non couvertes par les hypothèses de la preuve [Wagner 2004]. La sécurité de l'implémentation a donc été invalidée.

Nous introduisons différents modèles existants permettant de caractériser les fautes physiques sur un composant et montrons qu'ils sont peu adaptés à la vérification de propriétés de sécurité au niveau du code source. Nous introduisons ensuite notre propre modèle prenant en compte le point de vue du développeur et exprimé au niveau du code source C d'un projet embarqué. Nous

montrons comment ce modèle peut servir pour transcrire de manière complète les effets possibles d'une attaque physique sur le code source et en déduire les impacts fonctionnels résultants. Ce modèle nous sert, dans les sections 4 et 5, à créer des méthodes permettant de vérifier le fonctionnement du code C sous attaques physiques. Ces méthodes passent particulièrement bien à l'échelle permettant de traiter des codes réels de carte à puce. La méthodologie purement logicielle établie permet de prédire en avance de phase les vulnérabilités possibles d'un code C face à des attaques physiques par simulation sur le code source. Cette simulation permet, sans changer les habitudes établies de développement et de test, de mettre en évidence les failles du code pouvant être exploitées par un attaquant.

3.2 Caractérisation de l'attaquant

Tel que nous l'avons vu dans l'expression des propriétés de sécurité, un attaquant est caractérisé par sa capacité à observer des éléments du système et en tirer une connaissance sur des informations sensibles utilisées lors de son exécution. Un attaquant peut aussi utiliser cette capacité d'observation afin d'apprendre comment fonctionne le système. Armé de cette connaissance, il peut agir sur ce système et en modifier le comportement à l'exécution. Cette capacité d'action est une seconde compétence qui caractérise un attaquant. Nous décrivons chacune de ces deux capacités.

3.2.1 Capacité d'observation

Un attaquant, suivant les moyens et les connaissances dont il dispose, est capable en observant un système embarqué lors de son exécution de déduire le fonctionnement interne de celui-ci. Il est éventuellement aussi en mesure d'observer directement les données manipulées. De telles capacités d'observation peuvent très facilement compromettre la confidentialité d'informations secrètes telles que les clefs de chiffrement utilisées lors de calculs cryptographiques par la carte.

Dans un système embarqué, les garanties en terme de sécurité et de confinement de l'information sont remises en cause par les canaux cachés. Dans la figure 3.1, un utilisateur ou attaquant du système a la possibilité d'observer non seulement les entrées et sorties du système mais aussi les fuites d'informations involontaires de celui-ci. Ainsi des informations sur les secrets contenus dans le système peuvent être transmis à l'extérieur par le biais de ces canaux auxiliaires. Ces canaux remettent en cause le cloisonnement des secrets à l'intérieur du système. Les fuites par ces canaux représentent les éléments observables par un attaquant et ont été symbolisés par la notation $Obs(C, \mathcal{T}_A)$ dans la définition 3 du chapitre 2. Cette notion a été définie comme l'ensemble des informations obtenues en observant le conteneur C dans l'hypothèse d'une attaque d'un certain type $\mathcal{T}_A$.

[De Haas 2007] donne de manière simple les principes physiques exploités dans des observations par canaux cachés ainsi que des exemples pratiques illustrant comment ils peuvent être utilisés pour compromettre un chiffrement DES et RSA. Ces exemples montrent qu'il est possible d'observer directement un secret lors de son utilisation en connaissant les signatures temporelles ou les profils de consommation de courant des instructions réalisées (si celles-ci sont distinctes). Ces exemples montrent aussi qu'en comparant des exécutions réalisées avec des entrées différentes (dont l'entrée valide) et en observant les fuites d'information obtenues par canaux cachés,

il est possible de déduire si l'entrée présentée est valide. En plus de ces exemples, les principales méthodes de sécurisation contre ces canaux cachés sont aussi abordées de manière théorique.

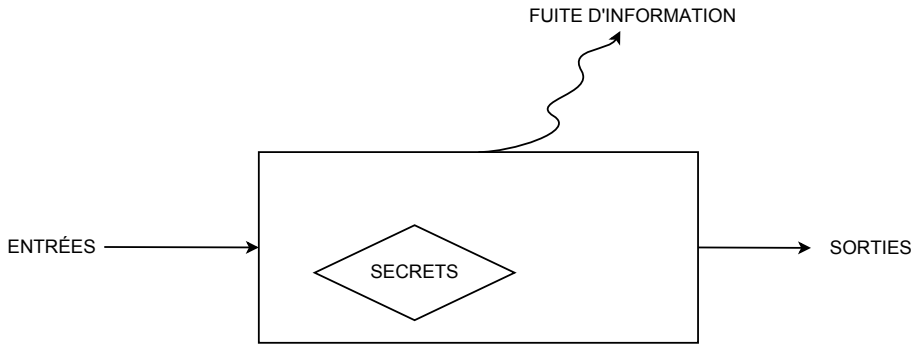


FIGURE 3.1 – Fuite d'informations par les canaux d'observation auxiliaires

Dans [Kocher 1996], les premières attaques par canaux cachés sont mises en œuvre sur des algorithmes cryptographiques en exploitant des informations acquises en mesurant les temps d'exécution des calculs effectués. Ces canaux cachés sont basés sur l'observation de fuites involontaires d'informations contenues dans les rayonnements physiques émis par le système lors de son fonctionnement. Différentes techniques d'observation et d'analyse sont maintenant employées. Outre la mesure du temps d'exécution, la mesure de la consommation de courant [Kocher *et al.* 2011] ainsi que la mesure de l'émission électromagnétique [Quisquater & Samyde 2001] sont utilisées. Des techniques d'analyse avancées utilisent une approche statistique basée sur la récolte et l'analyse par corrélation de multiples traces de consommation de courant obtenues au cours de l'exécution du code [Kocher *et al.* 2011].

Si de telles techniques avancées existent, les auteurs de [Kocher *et al.* 2011] affirment que des techniques d'observation simples sont suffisantes pour mettre en défaut des implémentations cryptographiques non protégées. Ces techniques ne sont pas nouvelles et doivent être considérées lors de la sécurisation d'un système embarqué. Les auteurs de [Akkar *et al.* 2000], discutent déjà de différentes techniques d'analyse de ce type et donnent une méthodologie pour réussir concrètement de telles attaques. Des résultats concrets d'attaques réussies sont obtenus dans [Gandolfi *et al.* 2001] où les principaux algorithmes cryptographiques comme le DES et le RSA sont mis en défaut. Ces résultats concrets ont poussé les auteurs de [Agrawal *et al.* 2002] à proposer un formalisme complet afin de modéliser les fuites passives d'information et à l'appliquer aux fuites d'information par rayonnement électromagnétique. Ce modèle appelé "adversarial model" repose sur un modèle mathématique basé sur la théorie de l'information. Il permet à l'aide des signaux de fuite observés combinés à une approximation statistique (gaussienne) de prédire si les signaux observés correspondent à une exécution connue prise sur un composant de référence. Si ce modèle prend en compte la présence d'un attaquant cherchant à obtenir la connaissance d'une information, il s'applique exclusivement aux signaux physiques observables lors d'une exécution et nécessite un apprentissage à l'aide d'un composant de référence et dans le cadre de fuites "passives". Ce modèle peut difficilement se traduire à un niveau de perspective proche du code source d'un programme et donc ne peut être utilisé tel quel par un développeur pour l'aider à sécuriser le code de son application.

Outre ces méthodes d'écoute passive, il existe des méthodes d'écoute active de type semi invasives ou invasives nécessitant souvent une préparation physique de la carte. Ces méthodes d'écoute active utilisent des techniques qui font intervenir une sonde électromagnétique ou de récupération [Samyde *et al.* 2002]. Les techniques de sondage permettent d'obtenir de l'information à l'aide des émissions électromagnétiques du composant et de déduire précisément quelle partie du composant est active au cours d'un calcul. Elles permettent également de retrouver de l'information par analyse de l'empreinte électrique laissée en mémoire par rémanence.

Pour plus d'informations sur le sujet des attaques par canaux cachés, une bibliographie a été compilée à la date d'octobre 2002 et a été publiée sous la direction de J.-J. Quisquater dans [Quisquater & Kouene 2002]. On y retrouve les principales méthodes décrites ci-dessus. Une liste de publications relative aux fuites d'information par canaux cachés et techniques associées est maintenue à l'adresse [SCL 2012].

Les techniques exposées ci-dessus montrent qu'un attaquant dispose d'une diversité de méthodes d'observation à sa disposition. Bien que ces méthodes nécessitent parfois du matériel sophistiqué, des plates-formes complètes sont maintenant disponibles [Riscure 2012] et réduisent l'investissement et la technique nécessaires pour observer les fuites physiques d'un système et en tirer avantage. Cependant l'expertise nécessaire pour exploiter les résultats fournis par ces bancs est encore élevée.

Les attaques mentionnés dans la littérature ciblent souvent les mécanismes cryptographiques du système embarqué. Cependant, le reste du code fonctionnel du système est lui aussi exposé aux mêmes attaques. La conclusion de [Giraud 2007] amène aussi à envisager les vulnérabilités du reste du système face aux attaques physiques. Pour soutenir cette affirmation, l'auteur de [Aigner & Oswald 2011] montre qu'il est possible par observation de la consommation de courant, de différencier deux instructions exécutées par le processeur. Dans son exemple, les `MOV 0x00` peuvent être différenciés des `MOV 0xFF` ainsi que les `JZ` des `JNZ` qui forment des traces de consommation bien distinctes si l'architecture matérielle du composant n'a pas été conçue pour ne pas émettre de fuites par rayonnement. On peut donc déduire qu'il est possible, pour un attaquant sur un système embarqué peu ou mal protégé, non seulement de comprendre le flot du code mais aussi d'obtenir des informations sur les données manipulées, et ceci par simple observation des canaux cachés. Le fait de connaître ces informations peut compromettre la sécurité du système en soi ou guider l'attaquant lors d'une tentative visant à perturber le système. Dans [Amiel *et al.* 2007] et [Clavier *et al.* 2010], les auteurs montrent la mise en œuvre des capacités d'observation afin de guider une perturbation physique du système.

Dans cette thèse, nous nous intéressons à la visibilité de l'information transmise directement à l'extérieur par les sorties prévues du système. L'observation par canaux auxiliaires peut cependant être incorporée dans le formalisme proposé en section 3.4 car celui-ci prend en compte les flux indirects ou partiels d'information.

Le but de cette thèse en rapport avec la fuite d'information se limite à proposer un formalisme décrivant les effets des attaques afin de vérifier si une attaque physique peut forcer une fuite d'information vers une sortie standard du système. A la différence des travaux cités précédemment, un tel modèle est formalisé au niveau du code source du logiciel considéré afin d'être au plus près de la terminologie du développeur. Il permet aussi une validation de propriétés de sécurité garantissant la confidentialité, l'intégrité et le bon fonctionnement de certaines parties critiques du code. Enfin, nous nous basons sur ce formalisme afin de proposer des solutions de vérification performantes.

3.2.2 Capacité d'action

S'il est possible d'observer le comportement d'un système embarqué pendant son utilisation avec les canaux cachés et d'en déduire des informations secrètes, le risque est d'autant plus important qu'un attaquant est en plus capable d'agir sur le système afin d'en modifier le comportement [Giraud & Thiebeauld 2004a]. Les techniques associées à ces modifications de comportement prennent la forme d'apport d'énergie tels que des surtensions hors des seuils de tolérance de la carte [Aumüller *et al.* 2002], des pointes de courant ou des modifications de la température agissant sur l'horloge de la carte [Anderson & Kuhn 1997], des émission de lumière [Skorobogatov & Anderson 2002] ou électromagnétiques [Quisquater & Samyde 2002]. Les premiers cas d'applications de ces attaques utilisées pour mettre en défaut des implémentations cryptographiques apparaissent dans les années 2000 [Biham & Shamir 1997] [Kömmerling & Kuhn 1999] [Boneh *et al.* 2001]. Cette capacité d'action alliée à la capacité d'observation utilisée comme guide ou comme moyen de compromission ouvre la porte à de nouvelles attaques. Deux études recensent les différents vecteurs possibles dont dispose un attaquant afin de perturber un système embarqué par une attaque physique : [Bar-El *et al.* 2006] et [Giraud & Thiebeauld 2004b]. Les attaques présentées ne demandent souvent que peu d'investissement et peuvent pour la plupart être effectuées sans détruire la carte ce qui les rend réalisables et particulièrement dangereuses.

Les effets de ces attaques peuvent être catégorisés suivant différents modèles d'effet de fautes connus. [Gadellaa 2005] et [Nguyen 2011] répertorient et expliquent ces modèles de fautes :

Bit flip La valeur du bit est inversée par rapport à la valeur qu'il aurait eu sans perturbation ;

Set et Reset La valeur du bit est forcée à une valeur fixe. On parle de set si la valeur est fixée à 1 et de reset si la valeur est fixée à 0 ;

Collage La valeur du bit est forcée à sa valeur précédente. On parle de collage à 1 si la valeur précédente était égale à 1 et collage à 0 sinon ;

Random La valeur du bit est forcée à 0 ou 1 avec une probabilité de 50 %.

Le lecteur peut trouver une liste récente et complète des attaques par fautes classifiées d'une façon explicite à la référence [Verbauwhede *et al.* 2011].

Ces fautes physiques utilisées comme vecteur d'attaque ont une focalisation, une durée temporelle [Quisquater & Kouene 2002] ainsi qu'une méthode d'analyse qui mène à l'exploitation de la vulnérabilité. Comme méthode d'exploitation, on peut mentionner l'attaque par *safe error* qui par comparaison d'exécutions du programme fauté et non fauté et la connaissance de la valeur fixée par la faute ainsi que le fait que la faute doit avoir un impact sur le résultat, permet, dans le cas d'une non réaction de la carte à l'attaque, de déduire la valeur de la case mémoire faussée. Cette déduction se fait sans connaissance préalable et ne nécessite qu'une observation externe des réactions du système. Une autre méthode d'exploitation est la recherche de collisions. Alors qu'il n'existait qu'un nombre défini d'entrées produisant une certaine sortie, l'introduction d'une faute augmente ce nombre d'entrées possibles pour cette sortie. De telles collisions permettent à un attaquant de présenter une entrée incorrecte et d'obtenir une sortie correcte. Les techniques d'exploitation ont constamment évolué au cours du temps [Rankl & Effing 2003] et ont contraint les fabricants de cartes à puce à adapter leur système en implémentant de nouvelles contre-mesures.

[Dutertre *et al.* 2010] donnent une explication des phénomènes physiques entrant en jeu dans une attaque physique laser ainsi qu'un récapitulatif des principaux modèles de fautes connus classés suivant les différents critères mentionnés ci-dessus. Le regroupement de ces modèles de faute permet de déterminer que les fautes considérées sont soit des inversions ou collages sur un bit, soit des random sur un octet. Pour notre modèle d'attaque, nous choisissons d'adopter un modèle de faute sur un octet qui nous semble plus réaliste dans la pratique.

Pour une explication des effets matériels des attaques par laser sur un composant embarqué, le lecteur peut se référer à [Canivet 2009]. L'auteur donne une caractérisation des effets des injections de fautes sur les éléments matériels du composant. Il propose également une classification des effets d'une faute physique en 5 catégories suite à des campagnes d'attaques par surtensions et tirs lasers : sans effet, faute silencieuse, faux positif, erreur détectée, erreur non détectée. Dans le chapitre 5, nous utilisons une classification qui se rapproche de celle-ci.

Dans la suite de cette thèse, nous nous intéressons à l'effet de ces attaques sans nous préoccuper de la technique pratique mise en œuvre pour la réaliser. Nous nous concentrons essentiellement sur les attaques optiques dont les effets ont été discutées en [Skorobogatov 2005] et qui sont toujours d'actualité [Skorobogatov 2010] où l'auteur recommande au niveau logiciel une vérification constante des accès mémoire. En effet, [Clavier 2007] mentionne dans sa thèse qu'il suffit d'une faute d'un bit sur le premier `xor` du calcul d'un AES pour mettre en défaut cet algorithme. L'auteur stipule également que la maîtrise d'une faute sur un bit est très difficile.

Les autres attaques documentées laissent cependant penser qu'il est possible, avec des attaques physiques, de compromettre le comportement du code au delà des accès mémoire. Ces différents types d'effets qui peuvent être obtenus à partir d'une attaque physique se traduisent en une modification du code. Les auteurs de [Anderson & Kuhn 1997] mentionnent qu'il est possible d'induire un saut conditionnel non prévu ou d'empêcher qu'un saut conditionnel prévu soit effectué. Ces techniques ont été utilisées de manière pratique dans le passé notamment pour mettre en défaut les systèmes de protection des chaînes télévisées payantes [McCormac *et al.* 1995] expliquant l'importance de les prendre en considération aussi bien au niveau de la conception matérielle [Guilley 2007] que de l'implémentation logicielle.

[Gadellaa 2005] qui travaille sur la Java Card fait un regroupement de l'ensemble des attaques classées par moyen de mise en œuvre, type d'effet, niveaux de compétence de l'attaquant et maîtrise nécessaire pour mener à bien les attaques. Il met également en relation leurs effets sur les variables au niveau logiciel. Il raffine notamment le modèle de faute en introduisant les quatre déclinaisons suivantes pour les erreurs sur un bit :

Erreurs de bit précis la faute affecte un seul bit du code ;

Erreurs de bit inconnu la faute affecte un seul bit, l'attaquant cible une variable mais ne contrôle pas précisément où celle-ci est affectée ;

Erreurs de bit inconnu sur des variables inconnues la faute modifie une seule variable mais l'attaquant ne sait pas quelle variable est affectée ;

Erreurs aléatoires la faute modifie un nombre quelconque de variables.

Pour plus d'informations sur les différents modèles de fautes et contre-mesures associées appliquées à un contexte cryptographique, le lecteur peut se référer à [Otto 2005].

Dans notre modèle, nous adoptons le même point de vue que le modèle de bit inconnu. Nous supposons qu'un attaquant est capable de modifier une variable connue à une valeur inconnue

en modifiant uniquement un bit. Cependant, nous élargissons ce modèle pour prendre en compte toute modification du code (incluant les modifications d'opcodes) affectant un seul octet.

Les fautes décrites dans ces modèles de fautes sont ensuite utilisées afin de créer des vulnérabilités dans le programme qui est interprété par le matériel. Ces vulnérabilités présentes dans le code source sont ensuite exploitées à un niveau fonctionnel pour monter une attaque fonctionnelle et tirer un gain de l'erreur introduite. Des attaques fonctionnelles compromettant la confidentialité d'un secret ou l'intégrité d'un objet sensible, en exploitant les conséquences d'attaques physiques, peuvent être trouvés à la référence [Lancia 2012]. L'existence de ces attaques confirme le besoin de s'intéresser aux effets de ces fautes à tous les niveaux du système et à comprendre leur influence sur le code source afin de s'en prémunir.

3.3 Modélisation des conséquences à haut niveau

Cette section décrit un modèle permettant de simuler au niveau C des attaques par injection de code. Avant de justifier en section 3.4 pourquoi ce modèle est pertinent et représente convenablement les attaques physiques, nous donnons le principe de ce modèle d'attaque au niveau C et nous l'illustrons sur l'exemple du listing 3.1. Ces attaques assembleur peuvent être regroupées en deux catégories : celles qui modifient la valeur d'une variable et celles qui induisent un saut dans le code.

Modification de variable

Une attaque classique peut modifier la valeur d'un registre, par exemple en EEPROM. Dans ce cas, une variable du programme est affectée au moment de son chargement depuis l'EEPROM. Afin de représenter l'effet d'une telle attaque, le modèle C proposé introduit une nouvelle instruction C qui affecte une valeur arbitraire à cette variable entre les lignes 3 et 4 comme dans le listing 3.2.

Listing 3.1– Exemple de code

```
int  func() {                          1
    ...                                2
    target_val = 0x00;                 3
    ...                                4
    ...                                5
    if (!security(target_val))         6
        secret = authorize();          7
    return  secret;                    8
}                                      9
```

Listing 3.2– Modification de la valeur d'une variable

```
int  func() {                          1
    ...                                2
    target_val = 0x00;                 3
    target_val =  0xF0;                4
    ...                                5
    if (!security(target_val))         6
        secret = authorize();          7
    return  secret;                    8
}                                      9
```

Ce type d'attaque, ayant pour conséquence la modification d'une variable, est la plus utilisée par les attaquants [Derouet 2007]. D'autres types d'attaques permettent d'obtenir un effet différent, comme décrit dans la section suivante.

Les attaques par saut

Perturber une instruction peut amener à modifier la nature d'une instruction assembleur. Nous donnons plus de détails sur l'effet d'une telle perturbation dans la section 3.4.3. Même si la maîtrise d'une telle attaque est vraiment difficile, un attaquant peut réussir à introduire un saut dans le code. Dans notre modèle d'attaque au niveau C, nous proposons de simuler l'attaque à

l'aide d'une instruction *goto*, comme dans le listing 3.3 où l'attaque est injectée entre les lignes 4 et 5.

La modification d'une instruction assembleur peut aussi introduire une instruction NOP, qui peut être intéressante afin d'éviter une condition. Néanmoins, éviter une ligne C est inclus dans le modèle d'attaque *goto*. L'exemple du listing 3.4 montre une ligne qui a été supprimée par une attaque ce qui peut être simulée en utilisant un *goto* jusqu'à la ligne 7.

Listing 3.3– Attaque par saut

```
1  int func() {
2      ...
3      target_val = 0x00;
4      ...
5      ... goto label;
6      if (!security(target_val))
7      label:
8          secret = authorize();
9      return  secret;
10 }
```

Listing 3.4– Attaque par non exécution

```
1  int func() {
2      ...
3      target_val = 0x00;
4      ...
5      ...
6      if (!security(target_val))
7
8          secret = authorize();
9      return  secret;
10 }
```

Maintenant que nous avons introduit le principe du modèle proposé, permettant de simuler au niveau du code source C les attaques physiques, nous proposons d'étudier finement et de manière exhaustive les conséquences d'une attaque sur le code assembleur. Nous souhaitons montrer qu'un nombre important d'attaques induites par faute physique peuvent se représenter à l'aide d'un modèle d'attaque au niveau C.

3.4 Modèle d'attaque physique à haut niveau

Dans cette section, nous décrivons les attaques physiques sur les cartes à puce et leurs effets afin de remonter vers l'effet produit au niveau C. Ainsi, nous montrons que le modèle proposé en section 3.3 est approprié pour simuler les attaques physiques au niveau du code source C.

Nous introduisons d'abord en section 3.4.1 ce qu'est une attaque par faute, ses caractéristiques et ses effets possibles. Nous expliquons aussi dans cette section pourquoi il est important de prendre en compte deux types d'attaques : les attaques par NOP et les attaques par saut 3.4.2. Dans les sections 3.4.3 et 3.4.4, nous détaillons les effets de ces attaques par faute sur le code et sur les données. Afin d'étudier leurs effets et les raccrocher à notre modèle d'attaque, nous proposons de différencier les effets sur les valeurs des variables des effets sur le contrôle de flot. Finalement, dans la section 3.4.5, nous abordons un aspect supplémentaire des effets des attaque physiques en expliquant comment une attaque par faute peut décaler le flot d'interprétation du code. Nous proposons également une modélisation de ce décalage et discutons des impacts possibles de celui-ci sur l'exécution du programme.

3.4.1 Effets des attaques physiques

Un attaquant peut modifier un registre ou la valeur d'une variable lors de son enregistrement dans une case mémoire en RAM ou EEPROM. Il peut perturber une variable pendant sa transmission sur le bus de données. En fonction du type de mémoire utilisé pour enregistrer la valeur attaquée et la potentielle répétition de l'attaque, l'effet de l'attaque est dit *transient* ou *permanent*.

Transient Une attaque transiente peut par exemple avoir lieu sur un bus lors de la récupération de l'instruction pour le processeur. Il n'y a pas de perturbation lors de la prochaine récupération de la même instruction. L'effet est donc ponctuel ;

Transient répétitif Une attaque transiente répétitive est une attaque transiente qui peut être effectuée plusieurs fois, ce qui a pour effet de provoquer le même effet à chaque fois qu'une même portion de code est sollicitée ;

Permanent Une attaque permanente a lieu quand un composant mémoire tel que l'EEPROM ou la RAM entre en jeu dans l'opération attaquée. L'effet perdure dans la case mémoire correspondante et se répercute sur les utilisations futures de cette case mémoire.

On appelle *attaque de donnée* une attaque qui modifie la valeur d'une donnée, i.e., une information du programme, résidant en RAM, EEPROM ou pendant sa transmission sur le bus de données. Ces attaques incluent aussi les données permanentes accessibles uniquement en lecture et donc stockées en ROM. On appelle *attaque de code*, une attaque qui modifie le code du programme manipulant ces informations. Ce code mémorisé en ROM peut aussi être attaqué pendant son transfert sur le bus de données. On distingue explicitement la cible d'une attaque (donnée, code) de l'effet de celle-ci (changement de la valeur de variables, changement de flot). Les attaques de code sont abordées dans la section 3.4.3 tandis que les attaques de données sont abordées en section 3.4.4. Les attaques de données et de code peuvent tous deux avoir des conséquences sur le fonctionnement du programme comme le montre la table 3.1. Ainsi, une attaque de code peut compromettre une opération arithmétique, ce qui a pour effet d'affecter une valeur erronée à une variable. Ce type d'attaque peut aussi compromettre une opération logique comme une condition, ce qui a une influence sur le contrôle de flot du programme. Les attaques de données, quant à elles, peuvent compromettre une information manipulées par le programme, c'est-à-dire, la valeur d'une variable. Cependant, si cette variable est utilisée dans une condition, de telles attaques auront aussi un effet sur le contrôle de flot du programme. Finalement, les deux types d'attaque peuvent tous deux avoir des effets à la fois sur les variables et le contrôle de flot du programme. Par la suite, nous détaillons les attaques de code dont les effets sont plus complexes comparés aux effets des attaques de données dont il est plus facile de comprendre qu'ils modifieront directement la valeur d'une variable.

Effet \ Cible	Code	Donnée
Variables	×	×
Flot	×	×

TABLE 3.1 – Croisement entre la cible et les effets d'une attaque

Nous avons choisi un modèle de faute qui nous semble réaliste parmi les différents modèles de fautes physiques possibles. Ce modèle de faute traduit les capacités de l'attaquant à modifier le système. Le modèle de faute retenu considère qu'il est possible pour l'attaquant de modifier K octets du code interprété mais qu'il est difficile pour lui de maîtriser la valeur obtenue sauf pour les valeurs `0x00` et `0xFF`. Soit X une instruction quelconque interprétée par le processeur à un instant donné de l'exécution du programme. Cette instruction X est transformée après l'attaque en une nouvelle instruction que l'on note Y.

$$X \overset{attaque}{\longrightarrow} Y$$

Or X et Y sont des instructions, donc on a la relation :

$$[\text{opcode, opérande1, opérande2}] \overset{attaque}{\longrightarrow} [\text{opcode', opérande1', opérande2'}]$$

Dans cette relation, pour le 8051, un opcode ou un opérande est codé sur un octet. Nous prendrons l'hypothèse qu'un attaquant est capable de modifier K octets dans une instruction modifiant un opcode, un ou plusieurs opérandes ou les deux. Nous supposerons aussi qu'une modification d'un octet s'effectue exactement sur un opcode ou un opérande mais pas "à cheval" entre les deux[1]. Si à cause d'une attaque, un opérande est interprété comme un opcode, un phénomène de décalage apparaît. Nous étudions ce phénomène en 3.4.5.

3.4.2 Importance des attaques par NOP et par saut

Dans cette section, nous détaillons les différents cas de figure possibles après une attaque et discutons de l'importance des attaques par NOP et par saut. Ces attaques sont de la forme :

$$X \overset{attaque}{\longrightarrow} Y$$

Plus précisément, dans les deux cas suivants :

$[\text{opcode, opérande1, opérande2}] \overset{attaque}{\longrightarrow} [\texttt{NOP}, \texttt{NOP}, \texttt{NOP}]$

$[\text{opcode, opérande1, opérande2}] \overset{attaque}{\longrightarrow} [\texttt{JMP}, \text{opérande1'}, \text{opérande2'}]$

Ces attaques peuvent être la conséquence d'une faute respectant le modèle d'attaque réaliste que nous avons retenu (perturbation d'un simple octet). Dans le document [Criteria 2009], de telles attaques ont une note *élevée* ce qui confirme leur importance. L'auteur de [Teuwen 2010] mentionne que les attaques les plus importantes sont les attaques où l'effet transforme Y en une instruction NOP ou une séquence de NOPs. Cette affirmation se justifie d'abord par le fait qu'obtenir une valeur précise entre `0x00` et `0xFF` est très difficile car elle dépend de la nature de l'attaque physique et de la disposition des composants matériels et de la réaction de ceux-ci. Ensuite, une telle attaque correspond à un pulse laser unique (hypothèse réaliste) par exemple sur le bus qui transmet le code devant être exécuté. Ces attaques sont en pratique les attaques les plus courantes car ces bus se trouvent à la séparation entre une zone mémoire et la "glue logique" du circuit ce qui les rend facilement distinguables. Enfin, dans la mesure du possible, un attaquant cherche à garder un contexte proche du contexte original en minimisant l'effet de l'attaque pour ne pas introduire d'effets de bord supplémentaires difficilement contrôlables. Cette dernière condition écarte les attaques sur un grand nombre d'octets consécutifs. Les attaques ayant des effets multiples seront décrites en section 3.4.3. Pour ces raisons, nous étudierons en plus d'une attaque générale, les attaques qui injectent une instruction de saut ou une séquence de NOPs.

1. En fait, une attaque "à cheval" sur une partie de l'opcode et une partie de son premier opérande revient à considérer que l'attaque modifie l'opcode et l'opérande.

Nous utiliserons comme exemple le listing de code 3.5 qui montre comment la perturbation d'un appel de fonction peut permettre la découverte du PIN au bout de 9999 essais au maximum.

Listing 3.5– Exemple d'une injection d'un NOP qui permet la découverte du PIN en 9999 essais

```
unsigned int user_tries = 0; // initialisation
unsigned int max_tries = 3;
// --vie de la carte--
ASM : CALL incr_tries(user_tries) -> NOP NOP
res = check_pin()
if (res == ok)
{ dec_tries(user_tries); }
if (user_tries < max_tries)
{ everything_is_fine(); }
else
{ killcard(); }
```

Dans la suite, nous discuterons de la façon de modéliser ces attaques pour les 3 sous cas suivants :

- les *attaques par* NOP qui remplacent une ou plusieurs instructions consécutives par des instructions NOP instructions ;
- les *attaques par* JUMP qui remplacent une instruction par une opération de saut, éventuellement arrière ;
- les *attaques générales* $X \overset{attaque}{\longrightarrow} Y$, i.e., le remplacement d'une instruction par une autre avec le même nombre d'opérandes.

3.4.3 Attaques de code

Une attaque de code modifie une instruction qui aurait dû être exécutée. Au niveau assembleur, cette modification est équivalente à une perturbation de l'opcode ou des opérandes d'une instruction ; éventuellement des deux. Quel que soit le jeu d'instructions de l'architecture cible, une instruction, une fois effectuée, mémorise des informations dans un ou plusieurs registres. Montrons qu'une attaque de code peut modifier le contenu d'un registre de destination, le contenu d'une case mémoire ou le contrôle de flot.

Les registres sont utilisés pour mémoriser des informations comme : des résultats intermédiaires lors d'un calcul, des adresses mémoire, des valeurs de variables ou des adresses d'instructions. Ces éléments doivent être utilisés (ce qui sous entend un accès par lecture) plus tard dans le code par au moins une instruction. Si ce n'est pas le cas, ces registres sont écrasés avant d'être lus (utilisés) – soit à cause du contrôle de flot soit parce que le compilateur a généré du code inutile – par conséquent une attaque sur ces registres avant cet écrasement n'a pas d'effet car l'écrasement annule l'effet de l'attaque.

L'utilisation du contenu d'un registre met à jour une variable (par une écriture en mémoire), calcule une adresse mémoire (par l'adresse d'une variable au cours d'un accès mémoire ou l'adresse d'un code exécutable lors d'un saut) ou influence le contrôle de flot (par un registre impliqué dans un saut conditionnel). Par conséquent, n'importe quelle attaque qui modifie le contenu d'un registre a soit un effet sur une mise à jour mémoire, soit sur le contrôle de flot, soit sur les deux.

Au niveau C, cela veut dire que n'importe quelle attaque qui réussit impacte une ou plusieurs variables, le contrôle de flot ou les deux. Dans les prochaines sections, nous montrons des exemples d'attaque au niveau assembleur afin d'illustrer ces différentes conséquences au niveau

Listing 3.6– Exemple d'une fonction C

```
char example(char u)
{
    char res, b;
    c = u + 5;
    b = c < 10;
    if (b) {
       res = c + 1;
    }
    else {
       res = 0;
    }
    return res;
}
```

Listing 3.7– Équivalent assembleur de la fonction exemple

```
_example:
   ...
   MOV  R2,DPL    // charge le paramètre en R2
   MOV  A,#0x05   // 5 est mis dans A
   ADD  A,R2      // compute u + 5 in A
   MOV  _c,A      // écrit c en RAM depuis A
   CLR  C         // baisse la carry
   SUBB A,#0x0A   // calcule b ,i.e., C-10
   JNC  00102$    // saute à 102 si la carry
                     n'est pas levée (else)
ATTACK_ADDR$:
   MOV  A,_c      // charge C dans A
   INC  A         // A++ i.e. c + 1
   MOV  R2,A      // écrit A en R2 (res=c + 1)
   SJMP 00103$    // saute par-dessus le else
00102$:
   MOV  R2,#0x00 // écrit 0 en R2 (res= 0)
00103$:
   MOV  DPL,R2    // pousse R2 sur la pile
   RET            // retour
```

FIGURE 3.2 – Exemple d'un code C et son équivalent en assembleur 8051C

C. Nous présentons également notre modèle d'attaque au niveau C pour chacune de ces conséquences. Nous prendrons d'abord comme exemple l'extrait de code C et l'assembleur équivalent présentés dans les listings 3.6 et 3.7.

Une approche similaire est adoptée dans un processus de rétro ingénierie ou en utilisant une technique comme la décompilation [Cifuentes & Gough 1995] où on cherche à retrouver, à partir d'un binaire destiné à être interprété par le matériel, le code source non compilé ayant servi à générer ce code binaire. Nous adoptons une approche similaire avec des hypothèses propres à un contexte embarqué sous attaques physiques afin de retrouver les effets au niveau fonctionnel d'une attaque physique malicieuse.

Dans la section 3.4.3, nous discutons des attaques qui impactent les données et proposerons une modélisation de leurs effets. Pour ces attaques, nous montrons plus tard dans le chapitre 5 que le modèle C équivalent est difficilement utilisable pour implémenter une plate-forme d'attaques exhaustives à cause du nombre total de cas à tester. Cependant, ce modèle sert de base pour nos contributions sur la vérification des propriétés d'intervalles sur les variables que nous présenterons dans le chapitre 4. Dans un deuxième temps, nous étudions dans la section 3.4.3 les attaques qui perturbent le code et leurs effets sur les valeurs des variables et le contrôle de flot. Nous proposons une modélisation de celles-ci et montrons comment le code source C peut être instrumenté pour les simuler dans le chapitre 5.

Les effets affectant les valeurs

Si une ou plusieurs instructions impliquées dans le calcul d'une nouvelle valeur pour la variable v est attaquée, alors la nouvelle valeur de v est faussée. Une telle attaque peut prendre la forme de la non exécution d'une ou plusieurs instructions (équivalente à des attaques par injection de NOPs ou un saut en avant dans le code), la répétition d'une ou plusieurs instructions

(équivalente à un saut en arrière dans le code) ou le remplacement d'une instruction par une autre. La variable v, affectée par l'attaque, est dite *attaquée* et nous appelons une telle attaque *une attaque de valeur* pour exprimer la modification de la valeur d'une variable. Une attaque de valeur a pour conséquence l'écriture d'une valeur faussée dans une case mémoire correspondant à une variable.

Au niveau C, une attaque par valeur peut être modélisée précisément si on sait quelle nouvelle opération est effectuée. Dans l'exemple de code en assembleur du listing 3.7, l'instruction `MOV A , #0x05` à la ligne 4 affecte la valeur $0x05$ au registre `A` afin de calculer `u + 5` avec l'instruction `ADD A, R2`, i.e., la nouvelle valeur de la variable c. Si l'opcode `ADD` à la ligne 5 est remplacée par un opcode `SUBB` alors l'affectation de c au niveau C devient `c = u - 5;`. Malheureusement, dans le cas d'une attaque, la valeur précise obtenue peut être difficile à modéliser au niveau C car il faut calculer la conséquence de l'attaque en tenant compte du code assembleur qui précède et succède le point d'attaque. Cela revient à "rétro ingénierer" du code assembleur vers son code source, si cela est possible. [Cifuentes & Gough 1995] adopte une telle approche de rétro ingénierie mais précise qu'elle dépend beaucoup des optimisations du compilateur. Or, si le compilateur est propriétaire, comme c'est souvent le cas dans un milieu industriel, une documentation complète des optimisations faites par celui-ci n'est pas forcement disponible. De plus, notre approche va plus loin que de la simple rétro ingénierie, puisqu'il s'agit de transformer le code binaire pour inclure une attaque avant de le "rétro ingénierer". De telles transformations sont utilisées dans [Griffith & Kaiser 2007] où les auteurs comparent leur propre outil d'instrumentation aux autres outils d'instrumentation pour la simulation de fautes existantes. Dans [Miller *et al.* 2001], les auteurs utilisent de telles techniques dans le but de vérifier la sécurité mais restent dans une optique de test en boîte noire sans remonter à l'origine de l'erreur.

A l'heure actuelle, la rétro ingénierie d'un calcul mettant en jeu des manipulations de registres peut être complexe à cause des optimisations de code introduites par le compilateur. Par exemple, le remplacement de `MOV A, #0x05` à la ligne 4 par une instruction NOP induit que l'opération `ADD A, R2` effectue l'addition entre les registres `A` et `R2` : le contenu du registre `A` est alors inconnu. Au niveau C, ceci est équivalent à `c = u + ?;`. Par conséquent, il est possible d'obtenir des valeurs qui peuvent compromettre la sécurité du système. Les auteurs de [Blömer *et al.* 2006] montrent qu'il est possible d'exploiter une telle attaque pour compromettre la sécurité d'un calcul cryptographique utilisant des courbes elliptiques. Dans leur mise en pratique, ils expliquent que n'importe quelle attaque menant à un changement de signe peut compromettre le calcul. Dans leur implémentation, le non branchement d'une condition est responsable du changement de signe, cependant ils mentionnent qu'une inversion de bit est suffisante dans d'autres implémentations du même calcul cryptographique [2].

Il existe ainsi de nombreux moyens de modifier le calcul d'une nouvelle valeur et il n'est pas réaliste de toutes les considérer. N'importe quelle attaque par injection de NOP sur les lignes 3, 4 et 5 ou des combinaisons de ces lignes perturbe le calcul de la nouvelle valeur de `c`. De même, n'importe quelle attaque par saut arrière depuis la ligne $i \geq 5$ jusqu'à la ligne 5 a pour conséquence un calcul d'une valeur erronée pour c et par conséquent représente une *attaque par valeur* sur `c`. Pour aller encore plus loin, si `MOV A, #0x05` est remplacé par une autre instruction avec le même nombre d'opérandes, par exemple `ADD A, #0x05`, une valeur inconnue pour la valeur

2. Cette différence renforce l'idée que la sécurité d'un système repose sur le couple de paramètres que sont la capacité de l'attaquant à modifier le système alliée à une implémentation spécifique.

Listing 3.8– Simulation 1

```
c = attack();          1
b = c < 10;            2
                       3
if (b) {               4
                       5
  res = c + 1;         6
}                      7
else {                 8
                       9
  res = 0;             10
}                      11
```

Listing 3.9– Simulation 2

```
c = u + 5;             1
b = c < 10;            2
goto label2;           3
if (b) {               4
                       5
  res = c + 1;         6
}                      7
else {                 8
label2:                9
  res = 0;             10
}                      11
```

Listing 3.10– Simulation 3

```
c = u + 5;             1
b = c < 10;            2
goto label4a;          3
if (b) {               4
label4a:               5
  res = c + 1;         6
}                      7
else {                 8
label4b:               9
  res = 0;             10
}                      11
```

Listing 3.11– Simulation 4

```
c = u + 5;             3
b = c < 10;            4
goto label4b;          5
if (b) {               6
label4a:               7
  res = c + 1;         8
}                      9
else {                 10
label4b:               11
  res = 0;             12
}                      13
```

`c` est calculée. C'est pourquoi nous proposons de modéliser n'importe quelle attaque modifiant la valeur d'une variable `v` par le *modèle d'attaque de valeur* suivant : `v = attack();`. Ainsi, tous les exemples d'attaque de valeur d'une variable `c` évoqués précédemment peuvent être modélisés dans notre modèle au niveau C de la même manière avec l'introduction d'une instruction `c = attack();` comme le montre le listing 3.8.

Arrivé ici, la difficulté principale est de tester toutes les variables attaquées pour toutes les valeurs possibles retournées par `attack()`. Au lieu de tester toutes les valeurs possibles pour une variable attaquée en utilisant un outil de simulation, il est plus approprié de prendre en compte ces attaques avec une approche statique. En effet, les outils statiques et sémantiques peuvent travailler sur des intervalles de valeurs pour les variables attaquées contrairement aux méthodologies de test qui utilisent des valeurs discrètes. Nous montrons comment ce modèle d'attaque peut être exploité dans le chapitre 4.

Les effets affectant le contrôle de flot

Illustrons les conséquences sur le contrôle de flot de nos 3 sous cas d'attaques : NOP, JUMP et $X \longrightarrow Y$.

Les attaques de contrôle de flot avec des attaques par NOP L'instruction assembleur `SUBB` calcule la valeur de la variable locale b à la ligne 8 du listing 3.7. Le registre de CARRY est positionnée à `1` par cette instruction quand il y a un dépassement, i.e., quand $c < 10$. Le registre de CARRY est utilisé plus tard par l'instruction de saut conditionnel `JNC` à la ligne 9. Par conséquent, une attaque par NOP ciblant l'instruction `SUBB` est une attaque de contrôle de flot par NOP. Puisque l'instruction `CLR C` à la ligne 7 positionne le registre de CARRY à `0`, si une attaque fait que la CARRY n'est jamais levée, l'instruction de saut conditionnel `JNC` à la ligne 9 introduit un saut qui est toujours emprunté. Une telle attaque peut être simulée au niveau C en insérant une instruction `goto` comme le montre le listing 3.9.

Supposons maintenant qu'une attaque par NOP cible à la fois les instructions assembleur `CLR C` et `SUBB A, #0x0A` aux lignes 7 et 8. Ce cas est aussi une attaque par contrôle de flot par NOP. Dans ce scénario, le saut conditionnel ligne 9 dépend de la dernière instruction qui a utilisé le registre de CARRY. Si la CARRY a été levée, le flot d'exécution du programme suit la branche `then` de la construction `if/then/else`, sinon le flot saute dans la branche `else`. Par conséquent, au niveau C, deux simulations sont à considérer, une pour chaque branche comme le montre

Listing 3.12– Simulation 5

```
goto label1;
c = u + 5;
label1:
b = c < 10;
if (b) {
  res = c + 1;
}
else {

  res = 0;
}
```

Listing 3.13– Simulation 6

```

c = u + 5;

b = c < 10;
if (b) {
label3: res = c + 1;
}
else {
  goto label3;
  res = 0;
}
```

Listing 3.14– Simulation 7

```
cpt ++;
c = u + 5;

b = c < 10;
if (b) {
  res = c + 1;
}
else {
label5:
  res = 0;
}
if (cpt >= first_time &&
    cpt <= last_time) {
goto label5; }
```

les listings 3.10 et 3.11. Notons que le modèle d'attaque couvre le cas où une attaque par NOP cible uniquement la ligne 8. En effet, une attaque modélisée au niveau C peut correspondre à plusieurs attaques au niveau assembleur qui peuvent être factorisée comme le montre les listings 3.9 and 3.11.

Maintenant, supposons qu'une attaque par NOP cible l'instruction assembleur `MOV _c, A` à la ligne 6 qui mémorise la nouvelle valeur de `c` en mémoire. Aucun registre n'est impacté dans ce cas, la variable `c` n'est juste pas mise à jour comme si la première instruction du code C n'était pas exécutée. Ce scénario peut être modélisé au niveau C avec le modèle d'attaque par GOTO en sautant la ligne C où `c` est affectée comme le montre le listing 3.12.

Par conséquent, toutes les attaques par NOP qui modifient uniquement le contrôle de flot peuvent être modélisées avec le modèle d'attaque par GOTO. Pour considérer l'intégralité des attaques de contrôle de flot par NOP au niveau C, il est alors suffisant de générer toutes les instructions GOTOs en avant dans le code d'une instruction C à une autre.

Les attaques de contrôle de flot avec des attaques par saut Nous considérons maintenant les attaques par saut, i.e., le remplacement d'une instruction assembleur par une instruction de saut. Comme les attaques par NOP sont équivalentes à des sauts en avant dans le code, nous ne considèrerons que les sauts en arrière dans cette section.

Supposons qu'une attaque par saut remplace l'instruction `MOV DPL, R2` à la ligne 19 par `JMP 00102$`, qui est un saut arrière dans le code (deux instructions en arrière) au label `00102$`, i.e., à la partie `else` de la condition. Cette partie affecte `0` au registre abritant la valeur de retour `R2`. Comme l'exécution est séquentielle, ce saut entraine une nouvelle exécution de l'instruction `JMP 00102$`. Si l'attaque est permanente, l'exécution du programme boucle de manière infinie. Si l'attaque est transiente ou transiente répétitive, la fonction s'achèvera et la valeur retournée est `0` (puisque la partie `else` est exécutée quelle que soit la valeur de `b` dans le cas d'une telle attaque). Ces deux scénarii peuvent être simulés au niveau C avec un GOTO. Dans le cas d'une attaque transiente ou transiente répétitive, le GOTO doit être encapsulé dans une condition comme le montre le listing 3.14.

Supposons maintenant que l'instruction `MOV R2, #0x00` est remplacée par l'instruction `JMP ATTACK_ADDR1$` où `@ATTACK_ADDR1$` désigne l'instruction à la ligne 10, la première de la partie `then`. Une telle attaque force le flot du code à emprunter la branche `then` depuis la branche `else`

(sans effectuer `res = 0;`). Par conséquent, si l'attaque est faite, le résultat de la fonction est `c + 1` au lieu de `0`. Cette attaque peut être simulée au niveau C par une instruction `goto` qui force le flot d'exécution à retourner en arrière dans la partie `then` comme le montre le listing 3.13.

On peut donc dire que tous les sauts arrières qui impactent seulement le contrôle de flot peuvent être modélisés avec un `goto` arrière au niveau C. Pour considérer toutes les attaques sur le contrôle de flot par saut arrière dans le code source, il suffit, au niveau C, de considérer tous les `goto` arrière d'une instruction C à une instruction précédente.

On peut conclure en disant que les attaques par NOP et par saut qui impactent seulement le contrôle de flot peuvent être modélisées avec le modèle d'attaque par GOTO au niveau C. Par conséquent, considérer toutes les attaques par GOTO au niveau C stimule l'ensemble de ces attaques.

Les attaques à multiples effets

Comme expliqué précédemment, une attaque ciblant un registre va se propager dans la suite du code jusqu'à ce que ce registre soit utilisé comme une variable (i.e., une attaque par valeur), ou pour contrôler le flot (i.e., une attaque de contrôle de flot). Cependant, suivant les optimisations réalisées par le compilateur, il se peut que ce registre soit utilisé plusieurs fois à la suite sans être réécrit à nouveau. Dans ce cas, en attaquant ce registre, l'attaque a plusieurs conséquences combinées. Déterminer si une attaque peut avoir plusieurs conséquences n'est pas aisé. Par exemple, le compilateur peut éventuellement extraire une sous expression d'un calcul arithmétique si le résultat de ce calcul intervient plusieurs fois par la suite, par exemple dans un autre calcul et dans des conditions. Cette sous expression peut alors être mise dans un registre qui est réutilisé plusieurs fois. Nous donnons dans le listing 3.15 un exemple simple de code ou `b+c` est utilisé à deux endroits. Si ce calcul est perturbé, la valeur de `a` est incorrecte et le `if` peut éventuellement mal se dérouler. Nos modèles d'attaque peuvent cependant représenter une telle attaque comme montré dans le listing 3.16 mais la difficulté pour détecter de tels cas demeure.

Listing 3.15– Code original

```
a = b + c + d; // attaque

if (b + c > 6) {
}
else {
}
```

Listing 3.16– Code avec attaques

```
a = attack();
goto label1; // ou goto label2
if (b + c > 6) {
label1: }
else {
label2: }
```

Considérons les attaques qui ciblent une des instructions impliquées dans le calcul de la variable locale `b` entrant en jeu dans la condition `if/then/else`. Une telle attaque peut modifier le flot du `if/then/else` et n'est pas une attaque par valeur puisque `b` reste dans un registre sans être stocké en mémoire. En fonction de la valeur du registre abritant `b`, l'attaque est équivalente à un saut inconditionnel dans la branche `then` ou la branche `else` du `if/then/else`.

On peut donc en déduire qu'attaquer une ou plusieurs instructions impliquées dans une décision de contrôle de flot et qui n'a pas d'impact sur une variable en mémoire est équivalent à l'insertion d'un saut inconditionnel dans le code. Nous appelons une telle attaque une *attaque de contrôle de flot*. Au niveau C, un saut inconditionnel peut être effectué avec un couple d'instructions `goto/label` où `label` désigne une instruction C cible pour le saut. Nous appelons ce modèle le *modèle d'attaque par* GOTO.

3.4.4 Attaques de données

Les attaques de données regroupent les attaques sur une information ou donnée manipulée par le programme. Un attaquant en modifiant les données manipulées par le programme à des moments clés peut changer la valeur d'une donnée sensible et par conséquent induire un nouveau comportement du programme. Les effets de ces attaques de données dépend de l'implémentation du programme et du gain visé par l'attaquant ; c'est pourquoi nous ne les détaillons pas. Leur modélisation est simple. Suivant le modèle de faute choisi, la modélisation revient à remplacer la valeur d'une variable lors de son utilisation (lecture ou écriture) par une autre dépendant des capacités de l'attaquant. Le modèle de faute que nous avons choisi induit que cette valeur est `0x00`, `0xFF`, ou une valeur aléatoire.

3.4.5 Décalage d'interprétation du code

Décalage du code

Nous avons vu que l'effet d'une faute physique pouvait introduire une modification de valeur de variables ou introduire des sauts modifiant le contrôle de flot du programme. Dans cette section, nous introduisons un phénomène supplémentaire causé par ces fautes physiques. Ce phénomène est associé à la perturbation du flot d'instructions interprété de manière séquentielle par le processeur. Nous cherchons à caractériser ce phénomène au niveau du code assembleur et d'en déduire qu'elles en sont les conséquences à plus haut niveau.

Une attaque qui modifie le code exécuté quand il est chargé depuis la ROM dans le CPU peut modifier la valeur d'un ou plusieurs octets. Dans le cas d'un jeu d'instructions de longueur variable, le processeur décode une instruction en commençant par décoder l'octet contenant l'opération code (opcode) pour ensuite déterminer le nombre d'octets nécessaires pour les opérandes. Ce mode opératoire est généralement le cas dans les systèmes embarqués de petite taille à cause du haut niveau de compacité du code résultant. Si une attaque modifie l'octet d'un opcode, elle peut alors aussi modifier le nombre d'octets suivants utilisés comme opérandes. Par conséquent, un décalage dans le flot d'instructions interprétées a lieu et ce flot peut rester décalé jusqu'à ce que le flot original soit éventuellement retrouvé.

Considérons l'exemple d'une attaque ciblant une instruction codée sur 3 octets [3]. Le premier octet X contient l'opcode tandis que les deux octets suivants sont les opérandes. Supposons que X est remplacé par Y. Les conséquences de ce changement dépend du nombre d'octets nécessaires par l'opcode Y :

1. X arg1 arg2 ; → Y ; arg1 arg2 si Y n'a pas d'octet d'opérande alors $arg1$ est vu comme un opcode. En fonction du nombre d'octets requis par $arg1$, $arg2$ est soit un opérande soit un opcode. Le flot d'instructions est décalé.
2. X arg1 arg2 ; → Y arg1 ; arg2 si Y requiert un octet d'opérande, alors $arg2$ est vu comme un opcode et le flot d'instructions est décalé (jusqu'à ce qu'il retrouve éventuellement le flot initial).
3. X arg1 arg2 ; →Y arg1 arg2 ; Si Y requiert deux octets opérandes alors l'intégralité de l'instruction est modifiée et le flot d'instructions n'est pas décalé.

3. D'autres cas avec un ou deux octets suivront le même schéma

Durée de vol

Lorsque le flot d'instruction est décalé, il peut éventuellement retrouver le flot initial, ce que nous étudions par la suite. Retomber sur le flot original consiste à interpréter comme opcode un opcode du flot original après un décalage d'interprétation induit par une attaque. Toute interprétation de code suite à ce décalage correspond à une étape. La première étape consiste à choisir un octet au hasard qui représente la premier octet lu comme un opcode après que le flot soit décalé. Cet octet, dans le flot original, peut être un opcode ou une opérande. Le fait de le choisir au hasard sert à représenter l'interprétation du processeur suite à une attaque décalant le flot d'interprétation. Toute étape suivante consiste à considérer l'octet courant comme un opcode et d'évaluer les octets suivants comme des arguments de cet opcode. Dans cette évaluation, on considère les sauts comme des instructions comme les autres, c'est-à-dire, que ce sont des opcodes ayant 0, 1 ou deux arguments suivant le saut considéré. On ne considère pas l'endroit où on va chercher l'instruction suivante mais considérons que le code est interprété de manière séquentielle. Cette partie peut se faire de manière expérimentale en "inlinant" le code qui doit être exécuté après le saut.

Quand le flot est décalé, le code binaire dans son intégralité peut être interprété comme un nouveau programme. Cependant, le programme original et le programme décalé peuvent correspondre à nouveau à un point donné : c'est ce que nous appelons *récupérer le flot original.* Avant de récupérer le flot original, un certain nombre d'opérations hors séquence peuvent être effectuées. Nous nommons l'ensemble de ces opérations hors séquence un *vol.* Ce vol se compose de plusieurs *étapes.* Une étape étant l'interprétation d'une instruction hors séquence par le processeur. Le but de cette section est de déterminer la *durée de vol moyenne*, définie à partir de la *durée de vol* :

Définition 14 (Durée de vol) *Nombre d'étapes nécessaires pour retrouver le flot original. Nous noterons $\bar{v}$, la durée de vol moyenne définie suivant un modèle probabiliste.*

On définit $p(n)$ comme la probabilité de retomber sur le flot original au bout de n étapes. Cette durée de vol moyenne se caractérise de la manière suivante :

$$\boxed{\bar{v} = \sum_{n=1}^{\infty} (n \times p(n))} \tag{3.1}$$

Cette notion de vol peut être illustrée par la figure 3.3. Les flèches supérieures représentent le flot initial tandis que les flèches inférieures représentent le vol, i.e., le flot décalé. Les cases vertes représentent les opcodes du flot d'origine. La case jaune représente la première étape de destination d'un vol c'est-à-dire un octet interprété comme un opcode hors de séquence originale. Dans cet exemple, le flot original est récupéré au bout d'un vol d'une durée de trois étapes. On considère que la première étape est causée par une attaque. Celle-ci contraint l'argument $A1$ à être interprété comme un opcode. Cet opcode nécessite deux arguments : $A2$ et $I2$. $A1$, $A2$ et $I2$ sont donc interprétés comme une instruction hors séquence formant une seconde étape dans le vol. Cette étape mène à l'étape $A3$, $I3$ interprété comme une instruction nécessitant un argument. Dans cet exemple informel, le flot original est finalement récupéré en $I4$ au bout de 3 étapes. Nous conserverons les conventions dans les schémas accompagnant les preuves.

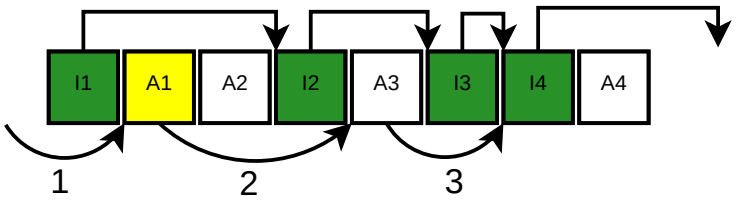


FIGURE 3.3 – Illustration de la notion de vol et des étapes

Modèle probabiliste

Afin de préciser la durée de vol moyenne dans un code, il faut définir quelques éléments complémentaires décrivant les distributions moyennes des instructions dans le code objet. En particulier, on définit, pour i le nombre d'arguments d'une instruction avec $i \in \{0, 1, 2\}$:

c_i **:** la probabilité que l'opcode du *code* interprété corresponde à une instruction nécessitant i arguments ;

a_i **:** la probabilité que l'opcode interprété fasse partie d'une instruction du *jeu d'instructions* nécessitant i arguments.

La deuxième série de probabilité dépend uniquement du jeu d'instructions que l'on considère, tandis que la première dépend du binaire considéré. On prend comme hypothèse que les arguments sont uniformément répartis sur l'intervalle [0..255]. Par conséquent, la répartition des *opcodes* correspondant aux arguments suit la loi de probabilité i.

Si on considère un octet au hasard dans un code binaire, cet octet O (en jaune dans les illustrations) est interprété comme un opcode dans la nouvelle séquence. On considère aussi que l'intégralité d'une instruction (opcode et un ou deux arguments) est toujours interprété de manière séquentielle. Les instructions elles-mêmes sont interprétées les unes à la suite des autres, ce qui implique qu'un saut dans le code est le résultat de l'interprétation d'une séquence d'octets représentant un saut dans le jeu d'instructions. Il n'est pas possible d'induire un saut autrement. Cette hypothèse exclut le cas où l'argument interprété n'est pas l'octet suivant directement un opcode ou le cas où un second argument n'est pas l'octet suivant directement le premier argument de l'instruction. Sachant qu'un opcode ou un argument sont codés sur un octet chacun, on a donc les probabilités suivantes s'appliquant sur le code original :

$p_0 = (c_0 + c_1 + c_2)/(c_0 + 2c_1 + 3c_2)$ de tomber sur un opcode ;

$p_1 = (c_1 + c_2)/(c_0 + 2c_1 + 3c_2)$ de tomber sur un argument juste avant un opcode ;

$p_2 = c_2/(c_0 + 2c_1 + 3c_2)$ de tomber sur un argument 2 octets avant un opcode.

Ces trois événements sont disjonctifs.

Dans ce modèle, nous prenons comme hypothèse qu'une attaque ne peut pas conduire le processeur à interpréter la moitié d'une instruction introduisant un décalage d'un demi octet dans le flot d'interprétation. Un tel phénomène nécessiterait une recherche en mémoire de l'instruction à interpréter d'une taille inférieure à un octet ce qui n'est pas le cas.

Lemme et preuve courte de la récupération du flot initial

Nous prouvons par la suite qu'un flot de code décalé a une forte probabilité de retrouver le flot initial. Le lemme suivant estime le nombre consécutif d'instructions exécutées par le processeur nécessaires pour récupérer le flot original. *Afin de simplifier le modèle, nous considérons dans un premier temps que le code assembleur est une distribution aléatoire du jeu d'instructions*, c'est-à-dire que $c_i = a_i$. Nous supposons aussi que choisir deux instructions consécutives dans le code binaire est équivalent à effectuer deux essais indépendants.

Lemme 1 *Quand le flot est décalé d'un octet, le nombre estimé d'instructions consécutives du processeur pour récupérer le flot original est* $1/p_0$.

Preuve: Une opération est constituée d'un opcode et de plusieurs opérandes (0, 1 ou 2 en 8051). Quand une attaque a lieu, le flot est décalé et ce nouveau flot d'instructions est interprété en utilisant un octet qui est un opcode du flot original avec une probabilité p_0 et un opérande avec une probabilité $1 - p_0$. Dans le premier cas, le flot original est retrouvé à la première étape. Dans le second cas, l'opérande (du code original) est interprété comme un opcode z. Le flot original peut être retrouvé à l'opération suivante en fonction du nombre d'opérandes de z et de la structure du flot original. Déterminer la nature de l'opération qui suit z peut être vu comme un autre décalage aléatoire indépendant du premier décalage. Par conséquent, la probabilité de récupérer le flot original reste p_0 à cette étape. Cette probabilité suit la loi de Bernoulli et nous permet de déterminer le nombre d'étapes $E(X)$ nécessaires afin de réussir à récupérer le flot original. $E(X) = p_0 + (1 - p_0)(1 + E(X))$ ce qui nous amène à $E(X) = \frac{1}{p_0}$. ■

Résultats numériques Pour le jeu d'instructions du 8051, en prenant l'hypothèse simplificatrice qu'un binaire suit une distribution uniforme des opcodes au sein du programme et en considérant que les opérandes suivent cette distribution, on obtient $p = 0.64$. Par conséquent, en 1.56 étapes le flot original est récupéré. Quand le flot d'instructions est décalé, la probabilité de succès de l'attaque responsable diminue car les opérandes qui sont interprétées comme des opcodes peuvent mener à un blocage du programme (si une instruction illégale est rencontrée ou si cette interprétation provoque une erreur fonctionnelle comme une division par 0). Cependant, de telles attaques ne sont pas impossibles à réaliser [Shacham 2007] et si le programme ne se bloque pas, le flot d'exécution récupère rapidement le flot original.

Théorème et preuve longue de la durée de vol

Théorème 1 *étant donnée le modèle probabiliste établi dans la section 3.4.5, la longueur du vol est :*

$$\bar{v} = p_0 + (1 - p_0)\left(1 + \frac{1}{\beta_1 + \beta_2}\right)$$

avec :

$$\beta_1 = \frac{c_1 + c_2}{c_1 + 2c_2}\left(a_0 + a_1 c_0 + a_2(c_0^2 + c_1)\right)$$

$$\beta_2 = \frac{c_2}{c_1 + 2c_2}(a_1 + a_2 c_0)$$

Preuve:

Détermination des cœfficients $p(n)$. Dans la suite, on détermine de manière précise les différents cœfficients de l'équation 3.1. Rappelons que $p(n)$ est la probabilité de retomber sur le flot original au bout de n étapes. Une étape est l'interprétation hors séquence d'un opcode. On considère que la première interprétation hors séquence représente déjà une étape comme on peut le voir dans la figure 3.3.

p(1) : Cette équation correspond à la probabilité de tomber directement sur un opcode original après une première étape et l'interprétation d'un opcode. Donc :

$$p(1) = p_0 = \frac{c_0 + c_1 + c_2}{c_0 + 2c_1 + 3c_2} \tag{3.2}$$

p(2) : Dans cette équation, l'octet considéré O est l'argument d'un autre opcode. Cette équation correspond à la probabilité de ne pas retrouver le flot original à l'issue de la première étape mais qu'une seconde étape permet de le rejoindre. On a :

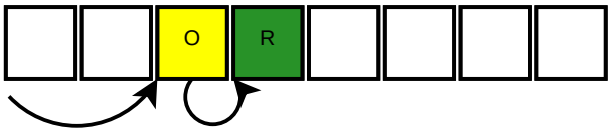
$$p(2) = (1 - p(1)) \times (cas(-1) + cas(-2)) \tag{3.3}$$

avec $cas(-i)$ représentant la probabilité conditionnelle de retrouver le flot en deux étapes sachant que l'on se trouvait à i octet(s) à gauche de l'opcode suivant. Ainsi $cas(-1)$ regroupe les cas où, à l'issue de la première étape, un argument du flot original interprété à cause du décalage comme un opcode se trouve à une distance d'un octet à gauche du prochain opcode du flot original. L'ensemble des cas peut être regroupé dans un des scénarii suivants.

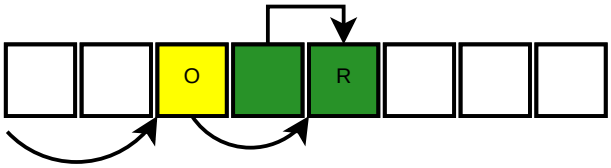
cas(-1) : Ce cas arrive avec la probabilité conditionnelle de $(c_1 + c_2)/(c_1 + 2c_2)$.

On détaille la contribution de ce cas de la manière suivante :

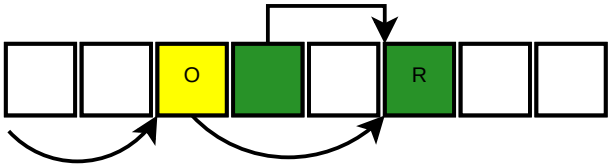
Avec probabilité a_0 : l'octet O considéré correspond à un opcode ayant 0 argument. On retrouve le flot à l'étape suivante ;



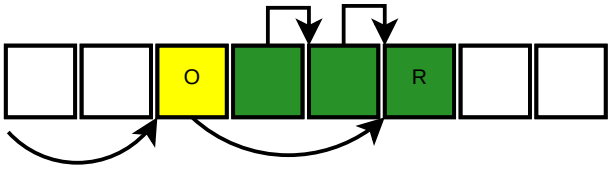
Avec probabilité a_1 : on retrouve le flot si l'instruction suivant O est une instruction ayant 0 arguments, c'est-à-dire avec probabilité c_0 ;



Avec probabilité a_2 : on retrouve le flot à l'étape suivante si l'instruction suivant O est une instruction avec 1 opérande, c'est-à-dire, avec probabilité c_1



ou si il existe deux instructions consécutives à O n'ayant aucun argument, cela correspondant à c_0^2.



La contribution pour le $cas(-1)$ est donc globalement de :

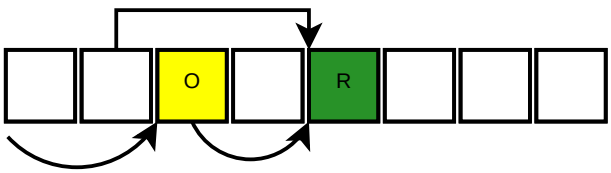
$$\boxed{\beta_1 = \frac{c_1 + c_2}{c_1 + 2c_2}\left(a_0 + a_1 c_0 + a_2(c_0^2 + c_1)\right)}$$

cas(-2) : la probabilité conditionnelle de ce cas est $c_2/(c_1 + 2c_2)$.

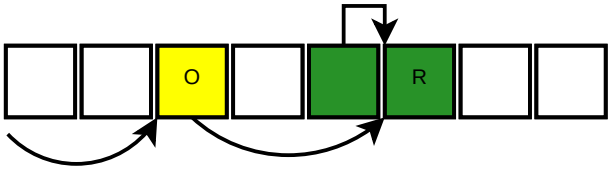
FIGURE 3.4 – Cas 2

On peut retrouver le flot à l'étape suivante dans les deux cas suivants :

Avec probabilité a_1 : l'octet O considéré est une instruction ayant 1 argument. Dans tous les cas, le flot est récupéré à cette étape



Avec probabilité a_2 : l'octet O considéré correspond à une instruction ayant deux paramètres. Pour récupérer le flot, il faut que l'instruction immédiatement suivante n'ait aucun argument. Donc, le flot est récupéré à cette étape avec la probabilité c_0



La contribution pour le $cas(-2)$ est donc :

$$\boxed{\beta_2 = \frac{c_2}{c_1 + 2c_2}(a_1 + a_2 c_0)}$$

Au final :

$$p(2) = (1 - p_0)(\beta_1 + \beta_2) \tag{3.4}$$

$p(n)$ **:** à une étape n donnée, il est assez simple de constater que l'on se retrouve dans un cas identique au précédent. On obtient donc le résultat suivant :

$$\boxed{p(n) = \left(1 - \sum_{j=0}^{i-1} p(j)\right)(\beta_1 + \beta_2)} \tag{3.5}$$

En résumé, à partir de la deuxième étape, cela correspond à un tirage suivant la loi de Bernoulli de paramètre $p = \beta_1 + \beta_2$.

L'espérance de cette partie est $1/(1-p)$ ■

Discussion du modèle et raffinement

Champs d'application Le modèle probabiliste employé prend en compte des étapes indépendantes (hypothèse du lemme 1) ou des étapes liées et interprétées de manière séquentielle (hypothèse du théorème 1). Or des instructions interprétées de manière séquentielle sans saut forment un bloc basique. Si on se place au niveau d'abstraction des blocs basiques, suivant les hypothèses adoptées, le lemme 1 s'applique sur les blocs basiques vus comme des étapes indépendantes tandis que le théorème 1 s'applique sur les instructions à l'intérieur d'un même bloc basique. La figure 3.5 illustre les champs d'application du lemme et de la preuve au niveau d'abstraction d'un bloc basique.

Raffinement Dans le calcul effectué dans le paragraphe 3.4.5, nous avons pris l'hypothèse que les instructions du jeu d'instructions étaient réparties uniformément dans le binaire ($a_i = c_i$). Dans la pratique, ce n'est pas le cas. On peut par exemple utiliser un jeu d'instructions dont la moitié des instructions sont des instructions illégales ce qui modifie totalement la dynamique du phénomène de décalage ainsi que le calcul du vol moyen. Il reste à prouver que le calcul du vol moyen repose uniquement sur la répartition des opérandes dans le code binaire et pas la répartition des opcodes et des opérandes. Le modèle peut donc être raffiné afin de s'adapter à des conditions différentes. Cependant le mode opératoire reste le même. De plus nous n'avons pas différencié la répartition des opcodes de celle des opérandes.

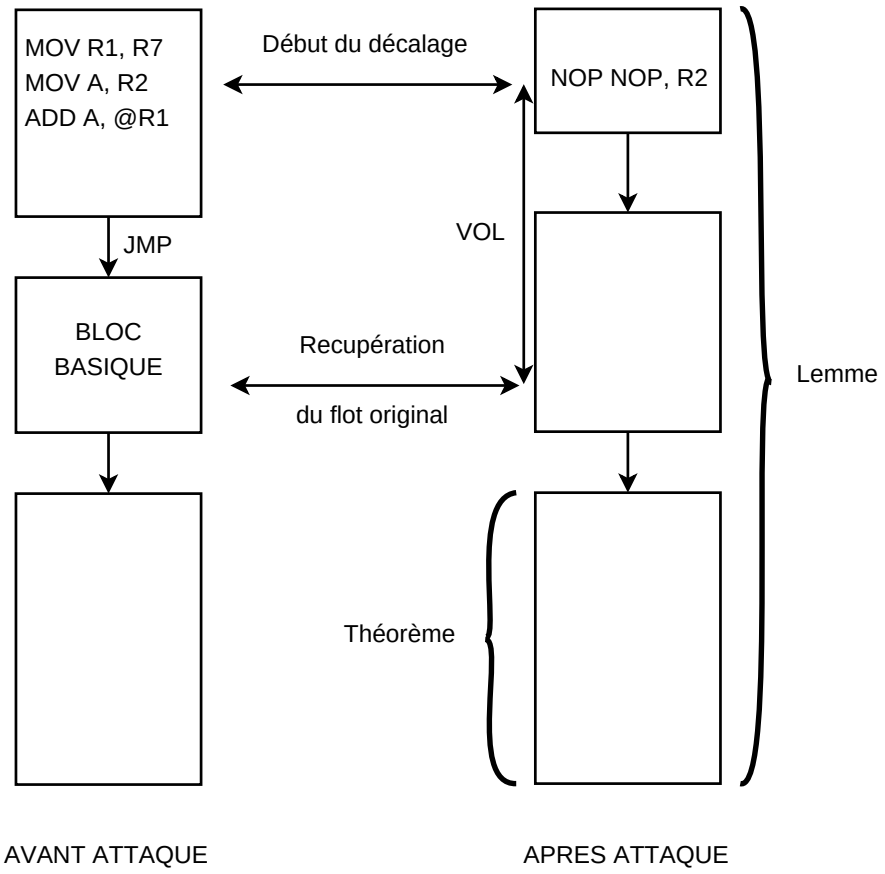


FIGURE 3.5 – Champ d'application du lemme et du théorème au niveau des blocs basiques

3.5 Conclusion

Dans ce chapitre, nous avons caractérisé de manière formelle les possibilités d'observation et d'action que nous considérons réalisables de la part d'un attaquant. Ces capacités vont conditionner la sécurité du système et les contre-mesures à rajouter dans le code. Nous avons décrit ces capacités d'une manière cohérente avec les propriétés de sécurité du chapitre 2, exprimées au niveau du code source pour être proche d'une représentation haut niveau propre au développeur. Nous avons ainsi défini un modèle d'attaque à haut niveau adapté à traduire les besoins fonctionnels du système et les garanties en sécurité qui en découlent. Ce modèle nous sert à caractériser les effets des attaques physiques sur le code source en essayant de faire apparaître les dangers fonctionnels qu'elles représentent. Nous pouvons maintenant utiliser cette caractérisation de l'attaquant et la confronter aux garanties de sécurité souhaitées pour notre système, exprimées au travers des propriétés de sécurité.

Ce modèle d'attaque permet dans un premier temps de vérifier si les propriétés de sécurité exprimées sont bien garanties. En caractérisant les effets d'une attaque au niveau du code source, le modèle permet de mieux comprendre celle-ci. Ce modèle inclut notamment une caractérisation du décalage d'interprétation de code résultat d'une attaque physique. On peut remarquer que contrairement à un modèle de faute un modèle d'attaque cherche à incorporer le

gain potentiel de l'attaquant. Le modèle ainsi défini permet aussi de modéliser les arrachements de la carte qui font partie des moyens dont dispose l'attaquant pour compromettre la sécurité [Marche & Rousset 2006]. En effet ces arrachements se résument à des sauts de code jusqu'à la fin du programme. Ce modèle permet donc de simuler des attaques afin de tester un système face aux attaques physiques. Les tests de sécurité réalisés nous permettent d'implémenter des contre-mesures appropriées aux endroits adéquats du système. Dans les chapitres suivants, nous proposons deux méthodes pratiques afin de vérifier la sécurité d'un système embarqué contre les attaques physiques. Ces méthodes reposent sur une analyse du code source et la simulation d'attaques physiques. Ces simulations d'attaques physiques sont établies en fonction du modèle d'attaque décrit dans ce chapitre.

3.6 Perspectives

3.6.1 Prise en compte d'attaques d'un ordre supérieur

Dans le modèle que nous avons établi dans ce chapitre, nous avons pris des hypothèses sur les capacités de l'attaquant ainsi que sur les conséquences matérielles réelles d'une perturbation physique. On peut facilement réaliser que la complexité de l'ensemble des interactions ayant lieu au niveau matériel ne peut pas être complètement traduite au niveau du code source et que seuls certains effets sont modélisables. Parmi ceux-ci, des effets supplémentaires d'une attaque peuvent être incorporés dans le modèle établi afin de l'étendre. Nous donnons quelques pistes afin de présenter ces conséquences multiples et les illustrer à l'aide d'exemples d'implémentations vulnérables.

Il est possible qu'une seule attaque ait des conséquences multiples. En effet, même une attaque simple peut avoir des conséquences modélisables par de multiples injections d'écriture et de multiples sauts. Par exemple, dans le listing 3.17, on peut voir qu'un seul saut a pour conséquence une modification de valeur et une modification de la logique du code par le remplacement d'un type de saut par un autre. Or ce remplacement de saut peut être simulé par une modification de la carry responsable du saut.

Listing 3.17– Exemple d'une attaque simple à conséquences multiple

```
MOV R1, 5                                  1
JNEQ label:12                              2
; devient après une attaque                3
MOV R1, 2                                  4
JEQ label:12                               5
```

Prenons comme exemple le code du listing 3.18 afin d'illustrer quelques exemples de vulnérabilités possibles en exploitant les conséquences multiples d'une attaque.

Listing 3.18– Code original

```
int foo;                                   1
foo = 1;                                   2
use_foo(foo);                              3
if (foo != 1)                              4
{                                          5
    KC();                                  6
}                                          7
```

Exemples de vulnérabilités De multiples conséquences peuvent être causées par des attaques doubles mais de simples attaques dans certaines conditions dépendant à la fois de l'implémentation du code, de l'architecture et du compilateur peuvent aussi provoquer des conséquences multiples sur le code. Ci-dessous quelques exemples permettent de se rendre compte de telles vulnérabilités.

Listing 3.19– Conséquence multiples : écriture + saut

```
int foo;                                                    1
foo = 1;                                                    2
foo = 2; // attaque modifiant foo                           3
use_foo(foo);                                               4
if (foo != 1)                                               5
{                                                           6
    // et provoquant aussi le saut de KC(C)                 7
    goto label_attaque;                                     8
    KC();                                                   9
    label_attaque:                                          10
}                                                           11
```

Dans le listing 3.19, si une attaque simple (ou une attaque double) a pour conséquence une modification de `foo` juste avant son utilisation en ligne 4 et un saut de la réponse sécuritaire `KC()` en ligne 9, `foo` est utilisé avec une valeur faussée sans que le code ne s'en aperçoive. Il s'agit ici d'une vulnérabilité.

Listing 3.20– Conséquence multiples : écriture + écriture

```
int foo;                                                    1
foo = 1;                                                    2
foo = 2;          // attaque modifiant foo                  3
use_foo(foo);                                               4
foo = 1;          // et modifiant foo une seconde fois      5
if (foo != 1)                                               6
{                                                           7
    KC();                                                   8
}                                                           9
```

Dans le listing 3.20, si une attaque (simple ou double) a pour conséquence deux modifications en écriture du code, on peut se retrouver dans un cas où la valeur utilisée par `use_foo()` est faussée et est remise à son état initial juste après son utilisation et avant sa vérification entrainant une réponse sécuritaire. Dans ce scénario, le code est une nouvelle fois incapable de détecter la présence d'une attaque. D'où la présence d'une vulnérabilité.

Ces deux exemples sont réalisables mais demandent une disposition particulière du code. Dans l'exemple suivant, nous choisissons une forme de code a priori sécuritaire et employée de manière courante dans les codes de cartes à puce.

Listing 3.21– Version sécurisée d'une condition vulnérable à une attaque par saut simple

```
if (x != 0)                                                 1
{                                                           2
    KC();                                                   3
} else                                                      4
{                                                           5
    code_effectif();                                        6
}                                                           7
```

L'extrait de code du listing 3.21, la condition `(x != 0)` prend en compte la majeure partie des valeurs possibles de `x` et les redirige vers la réponse sécuritaire `KC()`. Seule la valeur correcte

n'est pas filtrée et amène a l'exécution de `code_effectif()`. Cependant, cet extrait de code se traduit en assembleur de la manière suivante :

Listing 3.22– Traduction en assembleur du code C du listing 3.21

```
MOV R1, _x
MOV R2, 0
CMP R1, R2
JNEQ label_else
CALL KC()
JMP label_end
label_else:
CALL code_effectif()
label_end:
```

Or si ce code est attaqué par un simple saut de deux instructions, on se retrouve avec le code assembleur suivant :

Listing 3.23– Traduction en assembleur du code C du listing 3.21 sous attaque

```
MOV R1, #x
MOV R2, 0
CMP R1, R2
JNEQ label_else
GOTO label_else: ; attaque par saut sautant 2 instructions assembleur
~~CALL KC()~~
~~JMP label_end~~
label_else:
CALL code_effectif()
label_end:
```

On se retrouve donc à exécuter le code C du listing suivant, qui est exactement opposé en terme de logique à celui du listing 3.21 de départ.

Listing 3.24– Equivalent C du code assembleur attaqué

```
if (x != 0)
{
    code_effectif();
}
```

On peut ainsi voir que de telles attaques peuvent aussi être dangereuses pour la sécurité du système et doivent être prises en compte.

3.6.2 Extension du modèle

L'approche utilisée pour créer ce modèle repose essentiellement sur un critère de divulgation d'information ainsi qu'un aspect fonctionnel de la logique du programme. Une perspective pourrait être d'enrichir le modèle à l'aide d'une approche différente. [Pistoia *et al.* 2007] donne une idée d'une telle association entre le flot d'information et le contrôle d'accès.

Le modèle d'attaque établi cherche à lier des effets physiques bas niveau à des besoins fonctionnels de haut niveau afin de permettre au développeur d'associer les deux perspectives. Ces deux perspectives doivent être prises en compte afin de garantir la sécurité du système. Les besoins en sécurité s'expriment grâce à des propriétés de sécurité au même niveau que le modèle c'est-à-dire sur le code source du programme. Si cette approche a l'avantage d'être proche du point de vue du développeur, elle ne considère pas l'intégralité du système. Pour prendre en compte le système avec un niveau d'abstraction supérieur des modèles de menaces tels que

STRIDE peuvent être utilisés [Microsoft 2002]. Associer ces modèles de haut niveau permettrait de prendre en compte la sécurité du système de manière plus large et de considérer des vecteurs d'attaques différents.

Enfin, ce modèle ne considère que les attaques et ne modélise pas en soi les défenses. Combiner attaques et défenses dans une même représentation offrirait une solution intéressante permettant de les confronter. Les auteurs de [Kordy *et al.* 2010] proposent une telle approche sous la forme d'arbres d'attaque et de défense. Une telle représentation pourrait donner lieu à des formes de résolution utilisant, par exemple, la théorie des jeux [Manshaei *et al.* 2012]. Cette théorie a également été appliquée, au niveau du langage, à l'analyse de la sécurité du flot d'information d'un programme par interprétation abstraite [Malacaria & Hankin 1999].

Chapitre 4

Vérification de sécurité sous attaques simulées

Sommaire

4.1 Introduction . . . 109
4.2 Vérification statique . . . 110
4.2.1 Exemple de propriétés de sécurité . . . 111
4.2.2 Vérification d'intervalle de valeurs . . . 113
4.2.3 Limites de la vérification et solution . . . 115
4.2.4 Instrumentation et génération d'annotations . . . 115
4.2.5 Vérification de sécurité et simulation d'attaques physiques . . . 118
4.3 Réduction du nombre d'attaques simulées . . . 118
4.3.1 Attaques considérées . . . 120
4.3.2 Typage de code . . . 120
4.3.3 Classes d'équivalence . . . 122
4.3.4 Conditions et boucles . . . 124
4.3.5 Expérimentations . . . 137
4.4 Conclusion . . . 138
4.5 Perspectives . . . 139

4.1 Introduction

Dans ce chapitre, nous nous intéressons à la vérification automatique de propriétés de sécurité en nous appuyant sur une analyse statique du code C. Le langage C est utilisé pour le système d'exploitation des cartes à puce ainsi que pour ses applications. L'utilisation de ce langage bas niveau permet une meilleure efficacité par rapport à des solutions de type Java Card qui nécessitent une machine virtuelle. Néanmoins, le langage C souffre de différents inconvénients :

- Il n'est pas fortement typé : les outils d'analyse statique ont des difficultés pour l'inférence de type, en particulier avec les conversions de type et les pointeurs ;
- la mémoire peut être librement accédée : les outils d'analyse statique ne peuvent pas forcément déduire ce qui est manipulé sauf en représentant l'intégralité de la mémoire (avec des difficultés supplémentaires avec des structures de type union) ;
- le langage n'est pas orienté objet : les outils d'analyse statique ne peuvent s'aider de l'encapsulation ou de l'héritage pour en déduire des propriétés.

Ces inconvénients posent des difficultés aux analyseurs statiques et rendent plus difficile l'analyse de propriétés qui doivent prendre en compte ces restrictions. De plus, les codes de carte à puce contiennent des instructions assembleur qui manipulent directement des registres

et ne sont pas reconnus par les analyseurs statiques de code C se limitant le plus souvent au C de type ANSI. La justification derrière l'utilisation d'une analyse statique du code est qu'elle est indépendante de l'environnement ce qui la rend plus générique et lui permet d'être intégrée plus tôt dans le processus de développement de la carte. L'auteur de [Leveugle 2007] confirme l'intérêt d'intégrer des tests de sécurité de manière anticipée y compris au niveau matériel.

Dans ce chapitre, nous présentons une première méthode afin de vérifier que le code source respecte bien certaines particularités. Dans les chapitres précédents, nous avons présenté un formalisme permettant d'exprimer les besoins en sécurité d'un système et un modèle d'attaque servant à caractériser les possibilités d'un attaquant. Pour chacun d'eux, nous avons adopté une approche de haut niveau permettant de refléter les besoins fonctionnels du système sur le code source. Comme nous l'avons mentionné dans l'introduction, la carte agit passivement en réponse aux instructions du terminal. Ainsi, le code exécuté dépend essentiellement des informations se trouvant dans la commande envoyée à la carte et aussi des informations mémorisées en EEPROM. Ces informations seront manipulées par le programme à l'aide de variables au cours de conditions qui modifieront le comportement du programme. Afin de vérifier des besoins fonctionnels et leur sécurité, il convient de s'intéresser aux variables du programme et valider que les informations contenues dans celles-ci sont bien celles attendues même sous attaque physique. Si ce n'est pas le cas, un attaquant peut très bien à l'aide d'une attaque modifier le comportement du programme en sa faveur. Dans la section 4.2, nous présentons comment Frama-C, un outil d'analyse statique, peut permettre la vérification d'intervalles de valeurs dans des variables et introduisons par injection de code des simulations d'attaques physique. Dans la section 4.3, nous améliorons cette méthode en nous aidant des résultats établis dans les chapitres 2 et 3. Cette amélioration introduit des classes d'équivalence pour les conséquences des attaques. Ces classes permettent de considérablement réduire le nombre d'attaques à effectuer tout en couvrant l'intégralité des attaques de notre modèle. Nous apportons dans cette section la preuve de cette couverture ainsi que la méthode permettant de déterminer la position d'un nombre minimum de points d'attaques permettant de garantir cette couverture. Les contributions couvertes dans ce chapitre font référence à la publication [Berthomé *et al.* 2010].

4.2 Vérification statique

Lorsqu'on tente d'utiliser un analyseur statique sur un code relativement complexe, on s'aperçoit assez rapidement que des propriétés simples ne peuvent être vérifiées directement : l'outil ne peut conclure du fait des limitations précédemment évoquées. Une difficulté additionnelle pour les analyseurs statiques est le manque de contexte d'exécution par rapport à une analyse dynamique. Par exemple, les informations contenues en EEPROM ne sont pas déductibles de manière statique si elles sont incorporées dans une phase de personnalisation effectuée par le terminal. Dans cette section, nous proposons une méthode permettant d'aider l'outil de vérification à conclure sur le respect ou la violation de la propriété exprimée. Pour simplifier nos explications, nous n'avons considéré qu'une seule propriété concernant les variables : l'appartenance à un intervalle. Cela permet par exemple d'exprimer le fait que le PIN doit appartenir à l'intervalle de valeurs numériques $[0, ..., 9999]$.

Il existe deux avantages à vérifier statiquement des intervalles de valeurs. La première est d'ordre technique : elle permet en une seule passe de vérifier un ensemble discret de valeurs

qui demanderait dynamiquement autant de tests que de valeurs à tester. Sur des intervalles importants, le gain en terme de temps de test est significatif. L'autre avantage concerne la sécurité qui demande d'associer un sens à ces valeurs en fonction du contexte d'exécution. Ainsi, pour un scénario donné, il existe pour une variable du programme à un instant de l'exécution, un ensemble de valeurs qui peuvent être considérées comme bonnes ou mauvaises du point de vue de la sécurité. Cette différentiation dépend du gain que peut tirer l'attaquant à obtenir une valeur pour une variable différente de celle normalement attendue. Par exemple, réussir à réduire le montant d'une transaction à un montant inférieur à celui attendu peut lui être avantageux. Réussir à empêcher le décompte d'un compteur de sécurité, reflétant le nombre d'essais de présentation du PIN restants, constitue un autre exemple. En effet, cela permet à l'attaquant de découvrir le PIN à l'aide d'une présentation exhaustive d'un maximum de 9999 PINs. Dans ce dernier exemple, le fait de ne pas décompter se traduit par le fait que le compteur garde une valeur identique après décrémentation. Ces exemples peuvent donc se traduire en intervalles de valeurs ou un ensemble de valeurs discrètes. Ainsi, en vérifiant qu'une variable contient bien une valeur spécifique ou un ensemble de valeurs spécifiques à un moment de l'exécution du programme, on vérifie également que pour un scénario fonctionnel donné, le programme respecte bien le comportement attendu. De plus, on peut remarquer que vérifier qu'une valeur discrète appartient à un ensemble de valeurs défini revient à vérifier que cette valeur n'appartient pas à deux intervalles de valeurs (ou suite de valeurs discrètes continues). Ces intervalles sont définis comme l'intersection de l'ensemble avec la valeur discrète. On peut donc toujours créer un ou plusieurs intervalles de valeurs permettant de vérifier qu'une variable à un point donné du programme contient une valeur spécifique.

En partant d'un code de démonstration décrit dans la section 4.2.1, nous montrons que l'outil de vérification n'arrive pas à vérifier cette propriété d'intervalle et nous montrons comment injecter des informations complémentaires pour l'aider. Une fois cette étape franchie, nous présentons en section 4.2.5 comment prendre en compte les attaques physiques décrites en chapitre 3.

L'outil d'analyse statique retenu pour effectuer les analyses est Frama-C [Correnson *et al.* 2010]. Cet outil utilise des annotations afin d'exprimer les garanties souhaitées. Par conséquent, les objectifs de vérification doivent être exprimés dans le langage ACSL [Baudin *et al.* 2010]. Par exemple, l'intervalle de la variable `pin` s'écrit :

```
/*@ assert 0 <= pin_input <= 9999; */
```

4.2.1 Exemple de propriétés de sécurité

Soit l'extrait de code C présenté dans le listing 4.1. Ce code réalise une opération bancaire. La fonctionnalité principale se trouve dans la fonction `banking` qui appelle la fonction `work`. Ces deux parties de code sont extraites d'un code plus général écrit en C 8051. Cependant, les parties spécifiques au 8051 ont été remplacées par leur équivalent en C car elles sont difficilement interprétées par un analyseur de code source spécialisée dans le C ANSI.

Dans ce code, l'attaquant peut se concentrer en premier lieu sur les mécanismes d'authentification, en ligne 21. Pour un fonctionnement sans attaque, quand le mauvais code PIN est entré, la transaction est abandonnée. L'attaquant peut alors perturber la condition `if` pour obtenir malgré tout l'exécution de la fonction `work` pour tout code PIN entré. Dans le modèle d'attaque du chapitre 3, cette attaque peut être représentée à l'aide d'un `goto` injecté avant la ligne 22 et un label juste avant la ligne 23.

Listing 4.1– Exemple de code C pour une transaction bancaire

```
#include <stdio.h>                                                          1
typedef unsigned int word;                                                  2
word number1;           /* 2 variables globales initialisées ailleurs */    3
word number2;                                                               4
extern word scanp();    /* Méthode de saisie du code PIN */                 5
                                                                            6
word work() {           /* Travail une fois authentifié */                  7
    word result;                                                            8
    word x = 0;                                                             9
    if (number1 > 10)                                                       10
        x++;                                                                11
    result = number1 + number2 + x;                                         12
    return result;                                                          13
}                                                                           14
                                                                            15
void banking() {        /* Code principal de l'application bancaire */      16
                                                                            17
    word res;                                                               18
    word pin_input;                                                         19
                                                                            20
    pin_input = scanp();                                                    21
    if (pin_input == 1234) {                                                22
        res = work();                                                       23
    }                                                                       24
}                                                                           25
```

D'autre part, l'attaquant peut aussi se concentrer sur la fonction `work` qui calcule une valeur (par exemple le montant de la transaction) avant de retourner cette valeur. L'attaquant peut alors perturber un des termes de l'addition de la ligne 12. L'attaquant peut perturber par exemple la variable locale `x` ou l'une des variables globales *number*1 ou *number*2 afin d'obtenir le résultat souhaité. Par exemple, l'attaque `x=255;` peut être injectée juste avant la ligne 12. Pour obtenir ce résultat, plusieurs attaques sont possibles. Par ailleurs, si la fonction `work()` possède un nombre important d'instructions, cela donne plus de temps à l'attaquant pour réaliser l'attaque.

Les deux exemples d'attaques précédemment évoqués tentent de violer les propriétés de sécurité suivantes, comme décrites dans le chapitre 3 :

- "l'intégrité d'exécution de la ligne 22", afin de garantir la bonne authentification de l'utilisateur ;
- "l'intégrité de la variable result", censé garantir le bon calcul du résultat dans la fonction `work`.

Nous ne décrivons pas plus formellement ces propriétés en utilisant le formalisme du chapitre 3 puisqu'il est difficile de traduire de telles propriétés directement en ACSL. Cette partie mériterait une étude approfondie à part entière. Cependant, le langage ACSL permet d'exprimer des propriétés simples de comparaison de variables. Dans la suite, nous supposons donc par simplification que l'objectif de sécurité est de vérifier l'appartenance d'une variable à un intervalle. Comme évoqué précédemment, cela peut se traduire par la vérification que PIN appartient à l'intervalle $[0, ..., 9999]$.

Listing 4.2– Exemple de code C pour une transaction bancaire avec des annotations ACSL

```
#include <stdio.h>                                                          1
typedef unsigned int word;                                                  2
word number1;           /* 2 variables globales initialisées ailleurs */    3
word number2;                                                               4
extern word scanp();    /* Méthode de saisie du code PIN */                 5
                                                                            6
word work() {           /* Travail une fois authentifié */                  7
    word x = 0;                                                             8
    if (number1 > 10)                                                       9
        x++;                                                                10
    /* Check property 2 */                                                  11
    /*@ assert 0 <= x <= 1; */                                              12
    result = number1 + number2 + x;                                         13
    return result;                                                          14
}                                                                           15
                                                                            16
void banking() {        /* Code principal de l'application bancaire */      17
    word res;                                                               18
    word pin_input;                                                         19
                                                                            20
    pin_input = scanp();                                                    21
    /* Check property 1 */                                                  22
    /*@ assert 0 <= pin_input <= 9999; */                                   23
    if (pin_input == 1234) {                                                24
        res = work();                                                       25
    }                                                                       26
    /* Check property 3 */                                                  27
    /*@ assert 0 <= res <= 100; */                                          28
}                                                                           29
```

4.2.2 Vérification d'intervalle de valeurs

En considérant que `x` est une variable sensible, le développeur souhaite que cette variable appartienne à un intervalle prédéfini $[x_{min}, x_{max}]$. Le but de ce chapitre n'est pas seulement de vérifier qu'une telle propriété est satisfaite mais aussi qu'elle est garantie même si une attaque physique est réalisée. En effet, si elle est mise en défaut pour une attaque donnée, le développeur doit mettre en place une contre-mesure adéquate et doit ensuite vérifier à nouveau la robustesse de sa contre-mesure.

La propriété est exprimée en utilisant le langage ACSL [Baudin *et al.* 2010] dont les annotations sont incluses en tant que commentaires C. A partir de l'exemple du listing 4.1, on souhaite que les trois propriétés suivantes soient vérifiées.

1. Le `pin` doit être dans l'intervalle $[0, 9999]$ avant le test `if` (ligne 22) ;
2. La variable `x` doit être dans l'intervalle $[0, 1]$ avant de calculer `result` (ligne 12) ;
3. A la fin de la méthode `banking()`, le résultat doit être entre 0 et 100 (ligne 24).

On peut formaliser en ACSL ces trois propriétés à l'intérieur du code source, comme le montre le listing 4.2. Ces propriétés doivent être spécifiées par le développeur ou un spécialiste en charge de la sécurité de l'application.

La vérification de code source non attaqué incluant les annotations donne des résultats différents suivant la propriété considérée. Un extrait d'analyse Frama-C est présenté dans le listing 4.3. Frama-C isole d'abord les variables globales `number1` et `number2` et leur affecte un intervalle de valeurs inconnues. Ensuite, une analyse de valeur par interprétation

Listing 4.3– Analyse Frama-C du listing 4.2

```
[value] ====== INITIAL STATE COMPUTED ======                              1
[value] Values of globals at initialization                              2
        number1 ∈ [--..--]                                               3
        number2 ∈ [--..--]                                               4
[kernel] No code for function scanp, default assigns generated           5
banking-input2.c:23:[value] Assertion got status unknown                 6
banking-input2.c:12:[value] Assertion got status valid                   7
banking-input2.c:28:[value] Assertion got status unknown                 8
[value] ====== VALUES COMPUTED ======                                    9
[value] Values for function work:                                        10
          result ∈ [--..--]                                              11
          x  {0; 1; }                                                    12
[value] Values for function banking:                                     13
          res ∈ [--..--] or UNINITIALIZED                                14
          pin_input ∈ [0..9999]                                          15
```

abstraite [Cuoq & Prevosto 2010] est effectuée pour les variables et les valeurs obtenues sont confrontées aux exigences des annotations de type *assert*.

- La propriété 1 est testée à la ligne 23. Cette propriété ne peut être validée ($status = unknown$) car elle dépend de la fonction `scanp` [1] dont le code n'a pas été donné à Frama-C ;
- La propriété 2, à la ligne 12, est validée. Cette propriété est facilement validée par Frama-C ($status = valid$) car x est initialisé à zéro dans la fonction `work` et est incrémentée au plus une fois ;
- La propriété 3, à la ligne 28 ne peut être validée ($status = unknown$). Cela est dû au retour de la fonction `work()` qui retourne une valeur indéterminée car les valeurs des variables globales `number1` et `number2` ne sont pas connues.

On remarque qu'à la fin de l'analyse, Frama-C conclut que la variable `pin_input` est dans l'intervalle $[0, ..., 9999]$. Cela est dû au fait que chaque annotation ACSL est d'abord vérifiée et, si elle obtient le statut `unknown`, est alors supposée vraie pour la suite du processus de vérification. Ainsi, les résultats importants du listing 4.3 sont les validations d'assertion dans la partie *INITIAL STATE COMPUTED*.

Une autre remarque importante est qu'un `assert` est supposé vrai dès qu'il est de type `unknown`. Les vérifications suivantes utilisent alors la valeur supposée vraie pour cet `assert`. Ainsi, le prochain résultat obtenu dépend des résultats de type `unknown` qui le précèdent. Par exemple, la propriété 2, à la ligne 12, pourrait dépendre du résultat de la propriété 1, à la ligne 23, qui est de type `unknown` (i.e., supposée vraie pour la validation de la propriété 2). Cependant, dans cet exemple, il n'y a pas de dépendance entre les deux propriétés. Donc, pour que le résultat de l'ensemble des vérifications soit exploitable, il faut que toutes les propriétés soient déterminées (de statut `valid` ou `invalid`, mais pas `unknown`). Dans la suite, nous montrons une méthode pour aider l'analyse statique à déterminer les cas `unknown`.

1. `scanp` représente un équivalent de `scanf` pour la carte à puce : il lit le PIN depuis le terminal.

4.2.3 Limites de la vérification et solution

Les résultats observés dans le listing 4.3, nous ont montré, pour l'exemple de code contenant nos trois propriétés, que certaines propriétés ne peuvent être vérifiées par Frama-C. Plusieurs raisons expliquent cette limitation.

Premièrement, le code analysé est partiel. Le code de certaines fonctions, comme `scanp`, n'est pas donné à Frama-C. Cependant, même si le code relatif à `scanp` avait été fourni, aucun analyseur statique ne peut en déduire d'information exploitable d'une fonction qui demande des informations externes au programme de manière dynamique. Dans notre cas, les informations sont fournies dynamiquement par le terminal. Frama-C ne peut que supposer un ensemble de valeur maximal pour ces valeurs entrées, ce qui rend l'analyse indécidable. De plus, intégrer l'intégralité du code source d'un programme de carte à puce à l'analyseur est une opération longue et fastidieuse : le code comprend des appels non-ANSI et des portions de code assembleur qui perturbent l'analyseur. On cherche donc à restreindre le champ d'analyse afin de contourner ces difficultés.

Deuxièmement, Frama-C peut être perturbé par des codes C complexes, par exemple si des pointeurs de fonction C sont concernés par l'analyse. Ce problème, relatif aux analyseurs statiques, n'est pas un sujet traité dans cette thèse et n'est donc pas étudié plus avant.

Néanmoins, il est important de réussir à valider les propriétés de sécurité formalisées en ACSL pour le code standard, sans attaque. Pour ce faire, nous proposons dans la suite une méthode qui traite deux limitations précises dues à la structure du code :

- les variables globales qui peuvent être inconnues ou modifiées par des fonctions appelées dont l'analyseur n'a pas le code ;
- l'appel à des fonctions dont l'analyseur n'a pas le code et dont le retour définit une variable locale. Dans le code du listing 4.2, la fonction `scanp` illustre un tel cas.

4.2.4 Instrumentation et génération d'annotations

Dans cette section, nous proposons une solution pour aider l'analyseur statique lorsqu'il est en présence de variables globales et d'appels de fonctions externes. Nous proposons de collecter des informations à l'exécution pour ensuite les injecter sous forme d'annotations dans le code source de départ. Ainsi, de nouvelles informations seront disponibles lors de l'analyse statique. La solution est basée sur une instrumentation du code source afin d'observer l'évolution des variables globales et les résultats retournés par les fonctions durant leur exécution dynamique dans un scénario sans attaque. Afin d'obtenir des informations significatives, il est nécessaire d'exécuter un certain nombre de fois le programme dans des scénarii fonctionnellement cohérents. En effet, obtenir, pour une variable, des valeurs antagonistes du point de vue fonctionnel rend les valeurs obtenues incohérentes du point de vue d'un comportement de sécurité. Les conclusions élaborées à partir de l'analyse seront alors incorrectes. Cette instrumentation permet de construire des ensembles de données représentatives des exécutions observées. Pour une variable auditée, plusieurs cas sont possibles pour une variable observée :

- une même valeur est collectée à chaque exécution : on peut alors générer une annotation ACSL fixant la valeur de cette variable ;
- quelques valeurs sont collectées sur un nombre significatif d'exécutions : on peut alors générer une annotation ACSL décrivant un intervalle discret ou un intervalle si cela est

plus approprié ;

Une fois ces annotations générées, elles sont injectées dans le code original avant une nouvelle analyse statique des propriétés d'intervalles de sécurité voulues.

Code instrumenté pour l'observation

Un programme Java a été développé pour analyser les codes sources C et déterminer les instructions qui doivent être auditées. Dans notre cas, il s'agit de trouver les variables globales et les appels de fonction externes dont le résultat est utilisé (ou ayant un paramètre passé par adresse).

Par exemple, dans le listing 4.2, la valeur de la variable `number1` influence la condition ligne 9 et peut déclencher l'instruction `x++`. Pour chaque variable globale, le programme Java cherche les lignes de code utilisant cette variable en lecture. Juste avant chaque ligne de code, le programme injecte une séquence de code afin d'extraire la valeur de cette variable au moment de l'exécution dynamique du programme. Un exemple de séquence de code pour auditer la variable `number1` est présenté en listing 4.4. Il permet de récupérer la valeur de `number1` lors de son utilisation en ligne 9, ce qui explique la notation `@9`, qui permet ensuite de retrouver l'emplacement du point d'audit lors de l'analyse de la trace d'exécution.

Listing 4.4– Patch pour l'audit de la variable globale number1

```
/* DÉBUT AUDIT */                                                    1
char * buf_1 = ( char * ) malloc (20 * sizeof ( char ) ) ;          2
sprintf ( buf_1 , "%i" , number1 ) ;                                3
char temp2_1 [ 400 ] = "@9 : number1 = " ;                          4
strcat ( temp2_1 , buf_1 ) ;                                        5
fprintf ( dump , "%s\n" , temp2_1 ) ;                               6
/* DÉBUT AUDIT */                                                   7
if (number1 > 10)                                                   8
```

De manière similaire, le programme Java audite les fonctions non analysées par Frama-C. Pour `scanp`, la variable auditée est celle recevant l'affectation du retour de la fonction. La séquence de code d'audit est ajoutée juste après l'appel à `scanp`.

Après exécution, un fichier d'audit produit, par exemple, l'extraction suivante :

Listing 4.5– Extrait du fichier d'audit

```
@ 21 : pin_input = 1234                                             1
@ 9  : number1 = 10                                                 2
@ 13 : number1 = 10                                                 3
@ 13 : number2 = 0                                                  4
```

Injection des informations collectées

Les informations collectées pendant l'exécution sont lues dans le fichier de log et de nouvelles annotations ACSL sont insérées aux lignes fournies par ce fichier, comme le montre le listing 4.6, afin de permettre la vérification statique. Pour les variables globales, une annotation de type *assert* est ajoutée avant l'utilisation de la variable. Pour les fonctions inconnues comme `scanp`, une annotation de type *assert* est ajoutée après que la variable ait reçu l'affectation du retour de la fonction. Pour les ensembles discrets, l'opérateur logique `||` permet de représenter une énumération, comme par exemple `/*@ assert number1 == 10 || number1 == 11; */`.

Listing 4.6– Injection des annotations ACSL à partir des informations collectées

```
#include <stdio.h>                                                          1
typedef unsigned int word;                                                  2
word number1;           /* 2 variables globales initialisées ailleurs */    3
word number2;                                                               4
extern word scanp();    /* Méthode de saisie du code PIN */                 5
                                                                            6
word work() {           /* Travail une fois authentifié */                  7
    word result;                                                            8
    word x = 0;                                                             9
    /*@ assert number1 == 10; */        /* INSERTION AUDIT */               10
                                                                            11
    if (number1 > 10)                                                       11
        x++;                                                                12
    /* Check property 2 */                                                  13
    /*@ assert 0 <= x <= 1; */                                              14
    /*@ assert number1 == 10; */        /* INSERTION AUDIT */               15
    /*@ assert number2 == 0; */         /* INSERTION AUDIT */               15
    result = number1 + number2 + x;                                         15
    return result;                                                          16
}                                                                           17
                                                                            18
void banking() {        /* Code principal de l'application bancaire */      19
    word res;                                                               20
    word pin_input;                                                         21
                                                                            22
    pin_input = scanp();                                                    23
    /*@ assert pin_input == 1234; */    /* INSERTION AUDIT */               24
    /* Check property 1 */                                                  24
    /*@ assert 0 <= pin_input <= 9999; */                                   25
    if (pin_input == 1234) {                                                26
        res = work();                                                       27
    }                                                                       28
    /* Check property 3 */                                                  29
    /*@ assert 0 <= res <= 100; */                                          30
}                                                                           31
```

Listing 4.7– Analyse Frama-C avec les nouvelles informations

```
banking-input3.c:24:[value] Assertion got status unknown
banking-input3.c:25:[value] Assertion got status valid
banking-input3.c:10:[value] Assertion got status unknown
banking-input3.c:14:[value] Assertion got status valid
banking-input3.c:15:[value] Assertion got status valid
banking-input3.c:15:[value] Assertion got status unknown
banking-input3.c:30:[value] Assertion got status valid
[value] ====== VALUES COMPUTED ======
[value] Values for function work:
          result ∈ {10; }
          x ∈ {0; }
[value] Values for function banking:
          res ∈ {10; }
          pin_input ∈ {1234; }
```

Vérification des propriétés avec les informations collectées

Après avoir injecté de nouvelles assertions à partir de l'observation de l'exécution, Frama-C peut vérifier les propriétés demandées. La sortie de l'analyse est rapportée dans le listing 4.7. Les annotations de type *assert* qui correspondent aux informations injectées peuvent avoir un statut de type `unknown` car ces informations sont précisément ce que l'analyseur statique ne pouvait pas déduire dans la première analyse. Le fait que ces annotations soient de type `unknown` n'est pas gênant : ces assertions deviennent vraies pour l'analyseur statique. On observe que cela se produit bien pour les assertions des lignes 24, 10 et 15.

Ensuite, on observe dans le listing 4.7 que les propriétés visées obtiennent un statut `valid`. Cela se produit pour les trois propriétés, en ligne 25, 14 et 30. La collecte et l'injection d'informations a donc permis d'aider Frama-C à vérifier les propriétés voulues.

4.2.5 Vérification de sécurité et simulation d'attaques physiques

L'intérêt de vérifier des propriétés souhaitées sur le code original réside dans la vérification de ces mêmes propriétés en intégrant les attaques issues du modèle d'attaque présenté dans le chapitre 3. Dans la suite, on ne considère que les attaques de données, telles que définies dans la section 3.4.4 du chapitre 3. Le modèle d'attaque de contrôle de flot peut aussi être utilisé, cependant, traiter les attaques de données est particulièrement intéressant car un analyseur statique travaille justement sur les valeurs possibles des variables manipulées.

Soit une attaque physique perturbant une variable `x` juste avant la ligne 15 du listing 4.6 qui calcule le résultat de la fonction `work()`. L'attaque injectée peut par exemple être `x = 255`. Dans ce cas, la sortie donnée par Frama-C indique clairement que le statut de la troisième propriété est `invalid`, comme le montre le listing 4.8.

Ainsi, la méthode d'analyse statique proposée combinée au modèle d'attaque permet de mettre en évidence des violations des propriétés d'intervalle formalisées en ACSL.

4.3 Réduction du nombre d'attaques simulées

La section précédente a présenté une méthode pour vérifier une propriété de sécurité pour un code sous attaque physique. La difficulté pour exploiter une telle méthode réside dans le nombre

Listing 4.8– Analyse Frama-C avec une attaque de valeur simulée

```
banking-input3.c:24:[value] Assertion got status unknown
banking-input3.c:25:[value] Assertion got status valid
banking-input3.c:10:[value] Assertion got status unknown
banking-input3.c:14:[value] Assertion got status valid
banking-input3.c:15:[value] Assertion got status valid
banking-input3.c:15:[value] Assertion got status unknown
banking-input3.c:30:[value] Assertion got status invalid
(stopping propagation)..
banking-input3.c:43:[kernel] warning: non termination detected in function
banking
[value] ====== VALUES COMPUTED ======
[value] Values for function work:
          result ∈ {265; }
          x ∈ {255; }
[value] Values for function banking:
          NON TERMINATING FUNCTION
```

de codes différents à tester. Ce nombre dépend directement du nombre possible d'attaques physiques. Ce nombre est encore plus important si l'on souhaite simuler des attaques transientes ou transientes répétitives.

D'un point de vue spatial, l'attaque peut avoir lieu à tous les endroits du code source ce qui rend le nombre de codes sous attaque à considérer proportionnel à la taille du code. Ensuite, pour un point particulier du code, il faut considérer toutes les variables (resp., les contrôles de flot) qu'il est possible de perturber dans ce contexte. En se plaçant dans le pire des cas, l'ensemble des variables (resp., des contrôles de flot) du programme pourraient être attaquées. Ainsi, le nombre de codes à considérer est proportionnel à la taille du code multiplié par le nombre de variables (resp., le nombre de contrôles de flot). Afin de réduire le nombre de codes attaqués à tester, nous proposons une réduction qui minimise le nombre de codes attaqués à considérer. Le principe de l'approche utilisée est semblable à celle adoptée par [Barbosa *et al.* 2005] pour éviter la superposition de deux bit flips lors de tests d'injection de fautes. Nous utilisons une approche similaire au niveau C dans un contexte de sécurité pour réduire l'*espace d'attaque* à considérer.

Dans cette section, nous réduisons le nombre de codes en nous appuyant sur la notion d'équivalence de codes. Deux codes attaqués sont considérés équivalents quand les effets des deux injections donnent les mêmes conséquences lors de l'exécution du code. Par exemple, en considérant le code de la fonction `work()` du listing 4.2, l'injection d'une attaque contre `number2` a le même effet en ligne 8 ou en ligne 13. Ceci n'est pas vrai pour l'injection d'une attaque contre `number1` qui impacte la condition en ligne 9 si l'injection est en ligne 8. Ainsi, dans ce dernier exemple, il faut considérer deux classes d'équivalences pour les attaques concernant `number1` : les codes attaqués où l'attaque se produit avant la ligne 9 et les codes attaqués où l'attaque se produit après la ligne 9.

Dans la suite, nous utilisons les notions d'équivalence de codes et de couverture d'une classe d'équivalence par une autre. Le but est de calculer l'ensemble minimum de codes attaqués à donner à l'étape de vérification tout en garantissant la couverture totale de tous les effets d'attaques possibles. Les définitions et les algorithmes présentés dans cette section s'appuient sur la détermination des opérations de lecture et d'écriture pour des variables dont la modification peut potentiellement permettre de réaliser une attaque dangereuse pour l'application. Ces opérations de lecture et d'écriture correspondent aux définitions et utilisations des variables/don-

nées (use-def) et l'analyse de dépendance utilisée par les compilateurs pour effectuer certaines optimisations [Muchnick 1998].

4.3.1 Attaques considérées

Dans la suite de ce chapitre, nous supposons qu'une attaque permet de modifier une variable x. Typiquement, nous représenterons une telle attaque par le code $x = value$;.

Définition 15 *$Attack(code, i, x)$ est une copie du code original où l'attaque ciblant x est injectée entre les lignes $i-1$ et i.*

Nous considérons qu'un code source de taille n contient autant d'instructions C que le nombre de lignes, c'est-à-dire n. Ainsi, le code source résultant d'une injection d'attaque utilisant la fonction *Attack* possède $n+1$ lignes. Ce changement dans la numérotation du code source peut troubler les explications des sections suivantes. Nous considérons donc qu'une ligne injectée par l'attaque est insérée en ligne $i - \frac{1}{2}$ et n'a pas d'influence sur la numérotation du code source original.

4.3.2 Typage de code

A partir d'un code source *code* et d'une variable x, nous définissons les types *read* et *write* d'une ligne de code source en fonction de l'opération effectuée sur x. Dans certains cas, une ligne C peut effectuer les deux opérations simultanément, comme `x++` qui est typée *rw*.

Définition 16 *Soit $Type(code, x, i)$ défini comme suit : la ligne i du code est typée par read, write, rw ou $\emptyset$ pour une variable x. Le type read (respectivement, write) signifie que la ligne de code lit (respectivement, écrit) la variable x. Le type $\emptyset$ signifie qu'aucune opération n'est réalisée sur x.*

Dans la suite, nous raisonnons sur les lignes de code qui définissent une fonction f, notée $code_f$. L'analyse est locale au bloc de la fonction et le but est de déterminer la prochaine utilisation ou définition d'une variable, après qu'une attaque soit réalisée. Comme nous ne souhaitons pas analyser l'intégralité du code source, l'analyse s'intéresse uniquement au bloc de la fonction et n'entre pas dans l'analyse des appels de sous-fonctions.

Cependant, comme il est nécessaire de typer un appel de fonction appelé avec un paramètre sensible x, une analyse interprocédurale peut être effectuée afin de raffiner le typage des appels de fonctions. Par exemple, en fonction du corps de la fonction, un appel comme `proc(&x)` peut être typé à *read* ou *write*, ou bien $\emptyset$. La table 4.1 donne le typage des appels de fonctions que nous considérons et qui représente une sur-approximation de ces types qui est utile lorsqu'on ne possède pas le code source de ces fonctions ou que l'on ne souhaite pas les analyser.

La notion de typage d'une ligne de code peut maintenant être étendue à une définition récursive qui permet de qualifier, à partir d'un point particulier i le prochain type de x dans le code source :

Variable sensible : x	Si x est locale	Si x est globale
y = f(x) ;	read	rw
x = f(y) ;	write	rw
proc(& x) ;	rw	rw
proc(y) ;	$\emptyset$	rw

TABLE 4.1 – Règles de typage pour les appels de fonctions

Définition 17 *Soit $TypeAfter(code_f, x, i)$ la fonction qui retourne le type de la prochaine ligne typée (différent de $\emptyset$) à partir de i utilisant x. Elle est définie récursivement par :*

$$\begin{cases} TypeAfter(code_f, x, last) = \emptyset \\ TypeAfter(code_f, x, i) = \begin{vmatrix} Type(code_f, x, i) \text{ si } Type(code_f, x, i) \neq \emptyset \\ TypeAfter(code_f, x, i+1) \text{ sinon} \end{vmatrix} \end{cases}$$

où last est la dernière ligne du bloc de la fonction $code_f$.

$TypeAfter(code_f, x, i)$ ne retourne pas de type si la fonction n'utilise pas x après la ligne i, ligne i incluse. Si une ligne du bloc de code de la fonction utilise x après la ligne i (ou affecte potentiellement x dans l'un des cas de la table 4.1), $TypeAfter(code_f, x, i)$ retourne ce prochain type d'utilisation de x. Si ce type existe, nous pouvons extraire le numéro de la ligne qui définit ce type, comme le présente la définition suivante :

Définition 18 *Soit $LineTypeAfter(code_f, x, i)$ la fonction retournant le numéro de ligne définissant $TypeAfter(code_f, x, i)$, si ce type existe, - 1 a été choisi pour les autres cas.*

Algorithme 1 Calcul de TypeAfter pour un code séquentiel

```
1: function TypeAfter-sequentiel(code, x, i)
2:     if i = last(code) then
3:         return NULL
4:     Type type ← Type(code, x, i)
5:     if type ≠ NULL then
6:         return type
7:     return TypeAfter(code, x, i+1)
```

Dans le cas d'un code linéaire, ces deux fonctions peuvent se calculer par les algorithmes donnés aux algorithmes 1 et 2. Un code d'exemple est donné dans le listing 4.9 qui illustre les deux définitions précédentes.

Algorithme 2 Calcul de *LineTypeAfter* pour un code linéaire

```
1:  function LineTypeAfter-linéraire(code, x, i)
2:      Type type ← TypeAfter(code, x, i)
3:      if type ≠ NULL then
4:          j ← i
5:          while true do
6:              Type type2 ← Type(code, x, j)
7:              if type2 = type then
8:                  return j
9:              j+1
10:     return -1
```

Listing 4.9– Code d'exemple illustrant les fonctions de typage

```
word f() {                      // TypeAfter(x,1)=w                    1
    word result;                // Type(x,2)=∅                         2
    word y =  0;                                                       3
    word x = 0;                 // Type(x,4)=w                         4
    result = 0;                 // TypeAfter(x,5)=rw                   5
    x = x + 1;                                                         6
    result = result + y;        // TypeAfter(x,7)=r                    7
    y--;                                                               8
    send(x);                                                           9
    y--;                        // LineTypeAfter(x,10)=12              10
    result = 2 * y;                                                    11
    result = result + x;                                               12
    y = 0;                      // TypeAfter(x,13)=∅                   13
    return result;              // LineTypeAfter(x,14) n'est pas défini 14
}                                                                      15
```

4.3.3 Classes d'équivalence

Dans cette section, nous définissons la notion de classe d'équivalence sur les codes attaqués tels que définis dans la section 4.2.5. Nous restreignons pour l'instant l'étude à un code séquentiel au niveau du contrôle de flot, i.e., avec un code source qui ne contient pas de boucle ou de conditions : le code est alors vu comme une séquence linéaire d'instructions C comme des affectations, des appels de fonctions et des manipulations de variables. Pour un tel code, nous montrons que les classes d'équivalence correspondent au nombre minimal de codes à vérifier par l'analyseur statique pour garantir la propriété de sécurité voulue. Le cas général, pour un code contenant des conditions et des boucles, est étudié plus tard en section 4.3.4.

Définition 19 *Deux attaques* $Attack(code_f, i, x)$ *et* $Attack(code_f, j, x)$ *sont équivalentes, noté* $Attack(code_f, i, x) \equiv Attack(code_f, j, x)$*, si et seulement si les valeurs de* x *sont égales entre le code exécuté lors de* $Attack(code_f, i, x)$ *et le code exécuté lors de* $Attack(code_f, j, x)$ *pour chaque lecture de* x *pendant cette exécution.*

Il faut noter que l'on peut inclure le code original dans l'ensemble des codes attaqués. De cette manière, la classe d'équivalence de celui-ci correspond aux codes attaqués pour lesquels les attaques n'ont pas d'effet. Le lemme suivant permet d'identifier cette classe.

Lemme 2 *Si* $TypeAfter(code_f, x, i) \in \{write, \emptyset\}$ *alors* $Attack(code_f, i, x) \equiv code_f$.

Preuve: Dans le premier cas, $TypeAfter(code_f, x, i) = write$. La prochaine opération sur x après la ligne i est de type $write$, disons ligne j. Donc, il n'existe aucune opération $read$ entre les lignes i et j. Toute attaque qui modifie x dans cet intervalle est inutile car x est écrasé à la ligne j.

Dans le second cas, $TypeAfter(code_f, x, i) = \emptyset$ signifie qu'il n'y a aucune opération $read$ ou $write$ dans les opérations restantes. Donc, l'attaque n'a aucun effet. ■

Par exemple, en utilisant le listing 4.9, le lemme 2 implique qu'injecter une attaque contre x à la ligne 1 (avant l'exécution de la première instruction de la fonction) ou à la ligne 5 n'a pas d'effet sur le code : ces attaques sont équivalentes au code original.

Le lemme suivant permet d'identifier les autres classes pour un code linéaire :

Lemme 3 *Si* $TypeAfter(code_f, x, i) \in \{read, rw\}$ *alors*

$$Attack(code_f, i, x) \equiv Attack(code_f, LineTypeAfter(code_f, x, i), x).$$

Preuve: Après la ligne i, la prochaine opération concernant x est de type $read$, à la ligne $j = LineTypeAfter(code_f, x, i)$. Toute attaque qui modifie x dans l'intervalle $[i, j]$ a les mêmes conséquences sur x qu'une attaque effectuée en ligne j car aucune ligne dans l'intervalle $[i, j-1]$ n'utilise en lecture ou écriture x. Donc, tous ces codes attaqués sont équivalents au code attaqué en ligne j. ■

Par exemple, en utilisant le listing 4.9, le lemme 3 implique que l'injection d'une attaque contre x à la ligne 7 est équivalent à injecter l'attaque à la ligne 9. Ces deux lemmes mettent en évidence les attaques importantes qui ont des conséquences et celles qui n'en ont pas. L'idée principale est d'étudier les attaques qui ont des conséquences et de réduire leur nombre au minimum.

L'algorithme de la fonction $LineTypeAfter$ est relativement simple à mettre en œuvre à partir de l'algorithme linéaire. Le raisonnement est donné par l'algorithme 2.

Avec les deux lemmes précédents, on obtient un premier résultat pour les codes linéaires.

Théorème 2 *S'il y a n opérations de type read ou rw sur x dans* $code_f$ *alors le nombre minimal de classes d'équivalence est* $n + 1$ *et ces classes d'équivalence sont :*

$$code_f \cup \{Attack(code_f, i, x), \forall i \text{ t.q. } Type(code, x, i) \in \{read, rw\}\}$$

Preuve: Chaque ligne i du code est typé par la fonction $TypeAfter(code_f, x, i)$ par les valeurs $read$ ou $write$ ou rw ou $\emptyset$.

Si le type de cette ligne est $write$ ou $\emptyset$, le lemme 2 dit que le code attaqué est équivalent au code original et donc l'ensemble de ces attaques compte pour une classe d'équivalence : $code_f$.

Si le type de cette ligne est $read$ ou rw, le lemme 3 dit que ce code attaqué est équivalent au code attaqué à la ligne $j = LineTypeAfter(code_f, x, i)$. Cette ligne j définit donc cette classe d'équivalence. On remarque que cette ligne j est de type $read$ ou rw et que donc $Type(code, x, j) \in \{read, rw\}$. Réciproquement, deux lignes de type $read$ ou rw ont nécessairement deux valeurs de $LineTypeAfter(code_f, x, i)$ différentes et donc définissent deux classes

différentes. Donc chaque ligne définie par $Type(code, x, j) \in \{read, rw\}$ définit une classe unique. ■

Ainsi, il y a $n + 1$ codes à vérifier qui sont le code initial $code_f$, plus les codes attaqués où les attaques sont injectées aux emplacements d'opération *read* ou *write*.

Pour déterminer les classes d'équivalence de code qui contiennent des boucles ou des tests, la prochaine section doit reconsidérer les définitions des fonctions $TypeAfter$ et $LineTypeAfter$ qui devront prendre en compte les ruptures de flot.

4.3.4 Conditions et boucles

Dans la section précédente, nous avons seulement considéré les codes source sans boucles et sans conditions. Afin de traiter ces cas, les fonctions $TypeAfter$ et $LineTypeAfter$ peuvent être redéfinies afin de diviser le code d'une fonction contenant des boucles et des conditions en régions où les attaques sont équivalentes. Cependant, cela produit un grand nombre de classes d'équivalence. De plus, certaines classes peuvent être regroupées comme le montre le listing 4.10 en considérant des attaques contre la variable x. En utilisant la notion de classe précédente, on a deux régions : [1..3] et [4..7]. Si on divise ces classes à cause de l'instruction `if`, on obtient quatre zones : $[1, 2]$, $[3]$, $[4, 5]$ et $[6, 7]$.

Si on attaque en ligne 4 (attaque 1) ou en ligne 6 (attaque 2), si la condition est vraie, les conséquences de ces deux attaques sont identiques ; si la condition est fausse, l'attaque 1 est inopérante et l'attaque 2 a des conséquences sur la ligne 7. Dans tous les cas de figure, ces deux attaques ont des conséquences à la ligne 7 ou sont inopérantes : lors de l'analyse statique, elles peuvent être étudiées conjointement, en supposant que l'analyse du code original ait été effectuée au préalable.

D'un point de vue de notre modèle de classes d'équivalence, cet exemple définit trois classes :

- Le code original ;
- Le code attaqué en ligne 3 ;
- Le code attaqué en ligne 6.

Cependant, l'analyse statique du dernier cas est déjà traitée par l'analyse statique des deux cas précédents : on dit que cette attaque est couverte par les deux autres. Il est donc inutile de la simuler. Cette section fait évoluer les définitions des fonctions $TypeAfter$ et $LineTypeAfter$ pour tenir compte de cette situation.

Listing 4.10– Exemple de code contenant un if

```
...                                    1
if (cond){                             2
   z = x;                              3
   ← attaque 1                         4
}                                      5
← attaque 2                            6
y = x;                                 7
```

Définition 20 *Une attaque $Attack(code_f, i, x)$ est couverte par une attaque $Attack(code_f, j, x)$, noté $Attack(code_f, i, x) \lhd Attack(code_f, j, x)$, si et seulement si les effets de $Attack(code_f, i, x)$ sur les valeurs de x sont incluses dans les effets de $Attack(code_f, j, x)$ sur les valeurs de x ou si l'attaque $Attack(code_f, i, x)$ n'a pas d'effet.*

Dans la suite, nous généralisons les fonctions $TypeAfter$ et $LineTypeAfter$ pour des sections de code génériques. $LineTypeAfter(code, x, i)$ retourne un numéro de ligne tel que :

$$Attack(code_f, i, x) \lhd Attack(code_f, LineTypeAfter(code, x, i), x)$$

Ces deux fonctions sont étendues à tout bloc de C, i.e., entre des accolades équilibrées. Nous supposons que la numérotation des lignes est globale à tout le code source et que $first$ et $last$ sont deux fonctions sur un bloc C qui donnent respectivement les numéros de ligne du début et de fin d'un bloc considéré.

Conditions

Cette section s'intéresse aux conditions, en général les structures de type $if-then-else$. Une condition peut être décomposée en trois parties distinctes : la condition $cond$, le $then-block$ et le $else-block$. Ce dernier peut être vide. On définit les fonctions suivantes :

- $block-then(block,\ i)$ retourne le bloc "then" si une instruction `if` est située à la ligne numérotée i ou si i appartient à ce bloc "then". Dans le cas inverse, la fonction retourne NULL ;
- $block-else(block,\ i)$ est définie de manière similaire vis-à-vis du bloc $else$;
- $block-if(block,\ i)$ est définie de manière similaire pour tout le bloc if ;
- $first\ (block)$ retourne la première ligne du bloc $block$;
- $last\ (block)$ retourne la dernière ligne du bloc $block$. Si la conditionelle n'a pas de bloc $block-else$ alors $last\ (block-else)$ retourne la valeur de $last\ (block-then)$;
- $cond(if-block)$ retourne la condition du bloc $if-block$.

En utilisant ces fonctions, on redéfinit la fonction $TypeAfter$ dans l'algorithme 3. Le principe général de cette fonction est de trouver précisément l'utilisation suivante de la variable cible. Les différents cas sont explorés. Un cas particulier est quand on considère une ligne de code avant une condition et que la variable n'est pas utilisée jusqu'au `if`. Dans ce cas, si la variable est utilisée dans au moins un des blocs de la condition, le type retourné est l'union des deux types définis par chacun des blocs. Par exemple, si le bloc $then$ utilise la variable en lecture et le bloc $else$ en écriture, le type résultant est rw.

Le théorème 2 doit être modifié afin de prendre en compte les nouvelles définitions des fonctions $TypeAfter$ et $LineTypeAfter$. Avant d'arriver à la réécriture de ce théorème, nous devons adapter les lemmes 2 et 3.

Lemme 4 *Le lemme 2 reste valide dans le contexte des conditions.*

Preuve: On peut restreindre la preuve au cas des blocs if car la définition de la fonction $TypeAfter$ ne change pas hors de ce contexte.

Si $TypeAfter$ retourne $\emptyset$, cela signifie que l'analyse du code a traversé le bloc if sans trouver d'opération $read$ ou $write$. L'injection de code concernant cette variable est donc inopérante.

Si la fonction $TypeAfter$ retourne $write$, cela signifie que l'opération $write$ a été trouvée à l'intérieur ou après l'un des blocs du if. Deux cas sont alors à considérer.

1. Si l'opération $write$ est trouvée après le bloc if, alors l'opération annule toute modification de valeur par une attaque.

Algorithme 3 Fonction TypeAfter pour les conditions

```
1:  function TypeAfter(bloc, x, i)
2:      if i = last(block) then                                   ▷ Fin de bloc
3:          return ∅
4:      if Type(block, x, i) ≠ ∅ then          ▷ La variable est utilisée à cette ligne
5:          return Type(block, x, i)
6:      Block bif ← block-if(block, i)
7:      if bif ≠ NULL then                                ▷ La ligne est dans un bloc if
8:          Block bthen ← block-then(block, i)
9:          Type TypeThen ← TypeAfter(bthen, x, first(bthen))
10:         Block belse ← block-else(block, i)
11:         Type TypeElse ← TypeAfter(belse, x, first(belse))
12:         if i = line(cond(bif)) then                ▷ La ligne i est la condition du if
13:             if Type(block, x, line(cond(bif))) ≠ ∅ then
14:                 return Type(block, x, line(cond(bif)))
15:             if TypeThen = ∅ & TypeElse = ∅ then   ▷ x n'est pas utilisé dans then ni else
16:                 return TypeAfter(block, x, last(belse)+1)
17:             return TypeThen ∪ TypeElse ▷ renvoi de l'union des types des 2 branches sinon
18:     else if i < last(bthen) then                  ▷ La ligne i est dans le bloc then
19:         if ( thenTypeThen = ∅)                     ▷ et le bloc then n'utilise pas x
20:             return TypeAfter(block, x, last(belse)+1) ▷ recherche du type après la condition
21:         else
22:             return TypeThen
23:     else                                          ▷ La ligne i est dans le bloc else
24:         if TypeElse = ∅ then
25:             return TypeAfter(block, x, last(belse)+1)
26:         else
27:             return TypeElse
28:
29:     return TypeAfter(block, x, i+1)          ▷ On délègue le typage à la ligne suivante
```

Algorithme 4 Fonction LineTypeAfter pour les conditions

```
1:  function LineTypeAfter(bloc, x, i)
2:      if i= last(block) | TypeAfter(block, x, i) = write then    ▷ Fin de bloc ou écriture de x
3:          return -1
4:      Block bif ← block-if(block, i)
5:                          ▷ La variable est utilisée à cette ligne et on n'est pas dans un bloc if
6:      if Type(block, x, i) ≠ ∅ and bif = NULL then
7:          return i
8:
9:      if bif ≠ NULL then                                          ▷ La ligne est dans un bloc if
10:         Block bthen ← block-then(block, i)
11:         Type TypeThen ← TypeAfter(bthen, x, first(bthen))
12:         int lta-then ← LineTypeAfter(bthen, x, first(bthen))
13:         Block belse ← block-else(block, i)
14:         Type TypeElse ← TypeAfter(belse, x, first(belse))
15:         int lta-else ← LineTypeAfter(belse, x, first(else))
16:
17:         if i = line(cond(bif)) then                             ▷ La ligne i est la condition du if
18:             if Type(block, x, line(cond(bif))) ≠ ∅ then
19:                 return i
20:             if TypeThen = ∅ & TypeElse = ∅ then    ▷ x n'est pas utilisé dans then ni else
21:                 return LineTypeAfter(block, x, last(belse)+1)
22:                     ▷ sinon, x est utilisé au moins d'un côté en lecture, on "factorise" l'attaque
23:             return first(bif)
24:
25:         else if i < last(bthen) then                            ▷ La ligne i est dans le bloc then
26:             if TypeThen = ∅ then                                  ▷ et le bloc then n'utilise pas x
27:                 return LineTypeAfter(block, x, last(belse)+1)          ▷ on délègue à la suite
28:             else if lta-then ≥ i then
29:                 return line(cond)                                          ▷ On peut factoriser
30:             else                                ▷ x est utilisé au dessus : on ne peut pas factoriser
31:                 return LineTypeAfter(bthen, x, i)
32:
33:         else                                                    ▷ La ligne i est dans le bloc else
34:             if TypeElse = ∅ then
35:                 return LineTypeAfter(block, x, last(belse)+1)
36:             else if lta-else ≥ i then
37:                 return line(cond)
38:             else
39:                 return LineTypeAfter(belse, x, i)
40:
41:     return LineTypeAfter(block, x, i+1)          ▷ On délègue le typage à la ligne suivante
```

2. Si l'opération *write* est trouvée dans, par exemple, le bloc *then−block*, les conséquences d'une attaque peuvent être annulées par ce *write* ou bien être propagées par le bloc *else −block*. Dans ce dernier cas, cette attaque est considérée être couverte par des attaques injectées après le bloc *if*.

■

Deux opérations *read* peuvent se produire dans les blocs *then−block* et *else−block*. Pour couvrir toutes les attaques ayant un impact sur ces lectures, au lieu d'injecter une attaque dans chacun des blocs, nous pouvons injecter une attaque avant le début du bloc *if* si aucune opération sur x n'est réalisée entre la condition du *if* et ces deux lectures. Dans les autres cas, l'endroit où l'attaque doit être injectée est la ligne où la lecture est faite. Avec un tel raisonnement, nous pouvons redéfinir la fonction $LineTypeAfter(code, x, i)$ lorsqu'un *if* est mis en jeu.

Définition 21 *La fonction $LineTypeAfter(code, x, i)$, notée aussi $LTA(code, x, i)$, est définie par l'algorithme 4.*

Afin d'illustrer la fonction $LineTypeAfter$, nous montrons dans les listings 4.11 à 4.18 les différents cas de figure étudiés. Nous nous intéressons au placement des attaques en fonction de l'instruction `if`. Il faut examiner principalement deux cas : si l'attaque est placée avant ou dans l'un des blocs *then*/ *else*. Pour chacun de ces cas, on considère toutes les utilisations possibles de la variable cible x.

1. **L'attaque a lieu avant la condition**. Il faut remarquer que l'effet de cette attaque est le même que si l'attaque a lieu sur la condition.
   - La condition n'utilise pas la variable : elle est alors transparente pour le calcul de $LineTypeAfter$.

Listing 4.11– i en dehors de la condition — x non utilisé

```
 ← i
...
if ( ... )
{
    ...
}
else
{
    ...
}
 → LTA = LTA(11)
```

   - La variable est utilisée dans la condition. On peut placer l'attaque juste avant le `if` ; cette classe couvre aussi d'autres cas d'attaque décrits ci-après.

Listing 4.12– i en dehors de la condition — x dans condition

```
 ← i                                  1
...                                   2
 → LTA = 3 + 1/2                      3
if ( ..x.. )                          4
{                                     5
    ...                               6
}                                     7
else                                  8
{                                     9
    ...                               10
}                                     11
```

– La variable est utilisée dans le bloc *then* (symétriquement, dans le bloc *else*). On peut remonter cette attaque avant la condition, ce qui revient alors au cas précédent. Cette attaque n'a pas d'influence sur la partie *else* ; la déplacer est donc légitime.

Listing 4.13– i en dehors de la condition — x dans then

```
 ← i                                  1
...                                   2
 → LTA = 3 + 1/2                      3
if ( ... )                            4
{                                     5
    ..x..                             6
}                                     7
else                                  8
{                                     9
    ...                               10
}                                     11
```

– La variable est utilisée dans les deux blocs. On la factorise.

Listing 4.14– i en dehors de la condition — x dans then et else

```
 ← i                                  1
...                                   2
 → LTA = 3 + 1/2                      3
if ( ... )                            4
{                                     5
    ..x..                             6
}                                     7
else                                  8
{                                     9
    ..x..                             10
}                                     11
```

2. **L'attaque a lieu dans la condition**. On place l'attaque dans le bloc *then*, mais celle-ci peut être faite de manière symétrique dans le bloc *else*. Ce qui se passe dans le bloc *else* n'a pas d'importance pour ces attaques.

– La variable n'est pas utilisée dans le bloc. On peut déléguer le placement de l'attaque après le `if`.

Listing 4.15– i dans then — x non utilisé dans then

```
if ( ... )                                  1
{                                           2
 ← i                                        3
    ...                                     4
}                                           5
else                                        6
{                                           7
    ..x..                                   8
}                                           9
 → LTA = LTA(10)                            10
```

– La variable est utilisée avant et après l'attaque dans le bloc. On ne pas factoriser l'attaque ; cela revient au cas du code linéaire.

Listing 4.16– i dans then — x utilisé dans then avant et après i

```
if ( ... )                                  1
{                                           2
    ..x..                                   3
     ← i                                    4
    ..x..       → LTA = 5 + 1/2             5
}                                           6
else                                        7
{                                           8
    ..x..                                   9
}                                           10
```

– La variable est utilisée dans le bloc, après l'attaque, et il n'y a pas d'autre utilisation avant dans ce bloc. On peut remonter cette attaque avant la condition.

Listing 4.17– i dans then — x utilisé dans then après i

```
 → LTA = 1 + 1/2                            1
if ( ... )                                  2
{                                           3
     ← i                                    4
    ..x..                                   5
}                                           6
else                                        7
{                                           8
    ..x..                                   9
}                                           10
```

– La variable n'est pas utilisée après l'attaque dans le bloc. Ce cas revient au cas où la variable n'est pas utilisée dans le bloc (listing 4.15).

Listing 4.18– i dans then — x utilisé dans then avant i

```
if ( ... )                                  1
{                                           2
    ..x..                                   3
     ← i                                    4
}                                           5
else                                        6
{                                           7
    ..x..                                   8
}                                           9
 → LTA = LTA(10)                            10
```

Comme les attaques peuvent couvrir d'autres attaques, il faut adapter le lemme 3 qui exprime la couverture plutôt que l'équivalence de deux attaques qui ciblent une ligne dont le $TypeAfter$ est $read$ ou rw :

Lemme 5 *Si* $TypeAfter(code_f, x, i) \in \{read, rw\}$ *alors*

$$Attack(code_f, i, x) \lhd Attack(code_f, LineTypeAfter(code_f, x, i), x).$$

Preuve: Après la ligne i, la prochaine opération concernant x est de type $read$. Si la ligne déterminant le type n'est pas dans un `if`, on revient au raisonnement du lemme 3. La où les lignes définissant ce type peuvent être à plusieurs endroits : dans la condition, dans le bloc $then-block$, dans le block $else-block$, ou dans les deux. Plusieurs cas sont à distinguer :

- ce type n'est défini que par une seule ligne j :
  - de la condition du `if` : LTA renvoie cette ligne et toute attaque dans l'intervalle $[i, j]$ a les mêmes conséquences qu'une attaque effectuée juste avant la condition du `if` (listing 4.12).
  - du bloc $then-block$ ou $else-block$: LTA renvoie :
    - la ligne définissant le type (i est encadré par deux utilisations de x, comme montré dans le listing 4.16) : si la condition du `if` est vraie, l'attaque est couverte par la classe où l'injection a lieu juste avant la ligne j, ou sinon l'attaque est inopérante et donc couverte par la classe $code_f$.
    - la ligne de la condition : dans ce cas, il n'y a pas d'utilisation de x entre la condition et j (listing 4.17). Attaquer en ligne i est couvert par l'attaque effectuée sur la condition. En effet, si la condition est vraie, l'attaque est encore active à la ligne i et donc à la ligne j ; si la condition est fausse l'attaque est inopérante. Dans ce cas, l'attaque sur la condition couvre bien l'attaque en ligne i.
- ce type est défini par deux types $read$ ou rw en lignes j et j' : dans ce cas, la ligne i considérée est forcément avant le `if` et la ligne retournée par LTA est la ligne de la condition. Comme il y a deux lectures dans les blocs du `if`, on peut factoriser l'attaque à la ligne de la condition et donc couvrir toute attaque de $[first(block - then), j]$ ou $[first(block - else), j']$ par une attaque en ligne line(cond). Ce cas est illustré par le listing 4.14.

Ainsi, tous ces codes attaqués sont couverts par un seul et même code attaqué en positionnant le point d'attaque à la ligne $LineTypeAfter(code_f, x, i)$. ■

A des fins d'illustration du cas du `if`, un code d'exemple avec les valeurs de $TypeAfter$ (TA) et de $LineTypeAfter$ (LTA) est donné dans la table 4.2. Cet exemple montre la factorisation de la première lecture de x dans les deux branches de la condition. Pour les autres cas, la valeur de LTA est le numéro de la ligne où une lecture est effectuée. Pour la variable x, on s'aperçoit que LTA prend les deux valeurs 3 et 6 ce qui permet de définir deux classes de couverture pour l'ensemble des attaques contre x.

Boucles

Dans cette partie, nous modélisons la gestion des boucles. Seules les boucles `while` sont traitées puisque les autres types de boucles peuvent être réécrites avec un `while`. Les listings 4.19 à 4.25 illustrent les cas à distinguer quand un code contient une boucle. Ils montrent en particulier

L	Code	TA(x)	LTA(x)	TA(y)	LTA(y)	TA(z)	LTA(z)
1	word f()	r	3	r	3	rw	3
2	{	r	3	r	3	rw	3
3	if (y == 0) {	r	3	r	3	rw	3
4	y = 48 ;	r	3	w	-1	rw	3
5	y = x ;	r	3	w	-1	rw	3
6	z = x + z ;	r	6	∅	-1	rw	3
7	} else {	r	3	w	-1	∅	-1
8	y = x + 1 ;	r	3	w	-1	∅	-1
9	}	∅	-1	∅	-1	∅	-1
10	}	∅	-1	∅	-1	∅	-1

TABLE 4.2 – Exemple de valeurs pour la fonction $LineTypeAfter$ avec une condition

les lignes où les attaques doivent être injectées pour couvrir tous les cas d'attaque possibles. Nous distinguons deux zones d'attaque pouvant avoir une influence sur les instructions contenues dans une boucle : avant la boucle ou à l'intérieur de la boucle. Nous détaillons les cas possibles dans chacune des configurations.

1. **Avant la boucle :**

   – Listing 4.19 : la boucle n'utilise pas la variable x considérée ; elle est donc transparente pour les attaques. Les attaques dans ce cas seront couvertes par les attaques effectuées après la boucle.

Listing 4.19– i en dehors de la boucle — x non utilisé

```
 ← i                                  1
...                                   2
while ( ... )                         3
{                                     4
    ...                               5
}                                     6
 → LTA = LTA(7)                       7
```

   – Listing 4.20 : la boucle utilise la variable x au niveau de la condition. L'impact de l'attaque a lieu au moins sur la première évaluation de la condition et éventuellement sur les autres si x n'est pas modifié dans le corps du while. Cette attaque est couverte par l'attaque sur la ligne de la condition.

Listing 4.20– i en dehors de la boucle — x utilisé dans la condition

```
 ← i                                                    1
...                                                     2
 → LTA = 3 + 1/2                                        3
while ( ... x ... )// définit le type                   4
{                                                       5
    ...                                                 6
}                                                       7
```

– Listing 4.21 : la boucle utilise la variable au sein de la boucle. Une attaque à l'intérieur de la boucle n'est pas équivalente parce qu'elle est répétée à chaque tour de boucle. On peut donc, comme dans le listing 4.20, placer cette attaque avant la condition de la boucle.

Listing 4.21– i en dehors de la boucle — x utilisé dans la boucle

```
 ← i                                                    1
...                                                     2
 → LTA = 3 + 1/2                                        3
while ( ... )                                           4
{                                                       5
    ... x ... // définit le type                        6
}                                                       7
```

2. **A l'intérieur de la boucle :**

– Listing 4.22 : la boucle utilise la variable et l'attaque a lieu avant l'utilisation de celle-ci. L'impact de cette attaque a lieu à chaque tour de boucle au niveau de l'utilisation de la variable. Cette attaque est donc équivalente à celle effectuée sur la ligne d'utilisation de la variable ; elle est par conséquent couverte par cette dernière.

Listing 4.22– i dans la boucle — x utilisé dans la boucle après i

```
while ( ... )                                           1
{                                                       2
    ...                                                 3
     ← i                                                4
    ...                                                 5
     → LTA = 6 + 1/2                                    6
    x                                                   7
    ...                                                 8
}                                                       9
```

– Listing 4.23 : la boucle utilise la variable et l'attaque a lieu après la dernière utilisation de celle-ci. L'impact de cette attaque a lieu à chaque tour de boucle, mais seulement à partir de la deuxième. Elle peut ne pas avoir d'impact si la condition s'avère fausse. Cette attaque est donc équivalente à celle effectuée en fin de la boucle.

Listing 4.23– i dans la boucle — x utilisé dans la boucle avant i

```
while ( ... )                                    1
{                                                2
    ...                                          3
    x                                            4
    ...                                          5
     ← i                                         6
    ...                                          7
     → LTA = 8 + 1/2                             8
}                                                9
```

– Listing 4.24 : la boucle utilise la variable dans la condition et l'attaque a lieu dans le corps de la boucle. Cette attaque est effective à partir de la deuxième évaluation de la condition. La aussi, cette attaque est couverte par l'attaque effectuée en fin de boucle.

Listing 4.24– i dans la boucle — x utilisé dans la condition

```
while ( ... x ... )                              1
{                                                2
    ...                                          3
     ← i                                         4
    ...                                          5
     → LTA = 6 + 1/2                             6
}                                                7
```

– Listing 4.25 : la boucle n'utilise pas la variable. Dans ce cas, l'attaque dans le corps de la boucle n'a pas d'effet tant que l'on est dans la boucle. Elle est donc équivalente à une attaque effectuée juste après la boucle.

Listing 4.25– i dans la boucle — x non utilisé

```
while ( ... )                                    1
{                                                2
    ...                                          3
     ← i                                         4
    ...                                          5
}                                                6
 → LTA = LTA(7)                                  7
```

A partir de cette étude, les définitions des fonctions $TypeAfter$ et $LineTypeAfter$ peuvent être adaptées à la problématique des boucles. Pour cela, la ligne 28 de l'algorithme 3 doit être remplacée par l'algorithme 5. Pour réécrire une nouvelle fois le théorème, les lemmes 2 et 3 doivent rester valides. De manière similaire au cas de la condition, on définit de nouvelles fonctions :

– $block\_while(block,\ i)$ retourne le bloc "while", sans sa condition, si une instruction `while` est située à la ligne numérotée i ou si i appartient à ce bloc. Dans le cas inverse, la fonction retourne NULL ;
– La fonction `cond` est étendue pour prendre en compte la condition des boucles.

Lemme 6 *Avec cette modification de la définition de $TypeAfter$, le lemme 2 reste valide.*

Preuve: Comme la condition est évaluée après chaque partie interne d'un bloc `while`, toute attaque à la ligne i à l'intérieur d'un tel `while` peut toucher deux parties distinctes du code : premièrement, la partie juste après la ligne i, et deuxièmement la partie à partir de la condition

Algorithme 5 Code à ajouter à la définition de TypeAfter

Partie à insérer après la ligne 28 de l'algorithme 3

```
1:  Block bwhile ← block-while(block, i)
2:  if bwhile ≠ NULL then                          ▷ La ligne i est dans une boucle Type
3:      TypeEndLoop ← TypeAfter(bwhile, x, i)
4:      if TypeEndLoop ≠ ∅ then                    ▷ x est utilisée dans la boucle après i
5:          return TypeEndLoop
6:      Type TypeCondLoop ← Type(bwhile, x, line(cond(bwhile)))
7:      if TypeCondLoop ≠ ∅ then                   ▷ x est utilisée dans la condition
8:          return TypeCondLoop
9:      Type TypeBeginLoop ← TypeAfter(bwhile, x, first(bwhile)+1)
10:     if TypeBeginLoop ≠ ∅ then                  ▷ x est utilisée à partir du début de la boucle
11:         return TypeBeginLoop
12:     return TypeAfter(block, x, last(bwhile)+1)
```

du `while` jusqu'à la ligne i. Si la fonction TA retourne $\emptyset$, toute attaque n'a pas d'impact sur le reste du code. Si la fonction TA retourne *write*, une attaque est annulée par l'opération *write* dans la première partie de la boucle ou après le `while`. ■

Finalement, la définition de $LineTypeAfter$ doit prendre en compte de la même manière les boucles comme montré au listing 6. Il faut alors vérifier que le lemme 5 est toujours valide.

Avec cette nouvelle définition de la fonction $LineTypeAfter$, il faut vérifier que le lemme 5 est toujours valide pour prendre en compte la situation particulière des boucles.

Lemme 7 *Avec cette modification de la définition de $LineTypeAfter$, le lemme 5 reste valide.*

Preuve: Comme pour le lemme précédent, il faut s'attarder sur le cas des boucles. Plusieurs configurations peuvent se produire pour lesquelles le type de la prochaine utilisation est une lecture.

- **La prochaine utilisation est dans une condition de boucle.** Deux cas peuvent se produire :
  - La ligne i que l'on considère est avant la définition de la boucle. Cela revient au cas des conditions étudiées précédemment. La fonction $LineTypeAfter$ retourne alors la ligne de la condition de la boucle. Cette ligne est retournée par $LineTypeAfter$ dans ce cas par la ligne 5 de l'algorithme.
  - La ligne i est située dans le bloc while. Dans ce cas, la condition est évaluée avant de faire (ou pas) l'évaluation du bloc. Positionner l'attaque à la fin du bloc `while` permet que celle-ci soit effective lors de la prochaine évaluation de la condition (ligne 17 de l'algorithme).
- **La prochaine utilisation est en lecture et est à l'intérieur du bloc `while` en dehors de la condition**. À nouveau deux cas sont à considérer :
  - i est à l'extérieur de la boucle, placée avant. Dans ce cas, on ne peut pas mettre une attaque à l'intérieur de la boucle, car celle-ci est exécutée à chaque itération. La fonction $LineTypeAfter$ retourne la ligne de la condition de boucle (ligne 9 de l'algorithme) et cette attaque est couverte par celle où on attaque la ligne de la condition de la boucle.

Algorithme 6 Modifications à apporter à LineTypeAfter prenant en compte les boucles

Partie à insérer après la ligne 40 de l'algorithme LineTypeAfter

```
1:  Block bwhile ← block-while(block, i)
2:  if bwhile ≠ NULL then                                   ▷ i est dans une boucle
3:      Type TypeCondLoop ← Type(bwhile, x, line(cond(bwhile)))   ▷ i conditionne le while
4:      if TypeCondLoop ≠ ∅ & i = line(cond(bwhile)) then      ▷ et la condition est typée
5:          return i
6:      if TypeCondLoop = ∅ & i = line(cond(bwhile)) then      ▷ i conditionne le bloc while
7:          Type TypeEndLoop ← TypeAfter(bwhile, x, i)         ▷ et la condition est non typée
8:          if TypeEndLoop ≠ ∅ then   ▷ on regarde si x est utilisée dans le corps de la boucle
9:              return i
10:         else
11:             return LineTypeAfter(bwhile, x, last(bwhile)+1)
12:     Type TypeEndLoop ← TypeAfter(bwhile, x, i)     ▷ On est à l'intérieur d'un bloc while
13:     if TypeEndLoop ≠ ∅ then ▷ La variable est utilisée entre la ligne i et la fin de la boucle
14:         return LineTypeAfter(bwhile, x, i)
15:     Type TypeBeginLoop ← TypeAfter(bwhile, x, first(bwhile))   ▷ x utilisé dans le bloc ?
16:     if TypeBeginLoop ≠ ∅ then
17:         return last(bwhile)
18:     return LineTypeAfter(block, x, last(bwhile)+1)
```

- i est à l'intérieur de la boucle avant l'utilisation de x. Dans ce cas, l'attaque a lieu à chaque tour de boucle. L'effet de l'attaque à la ligne i est le même que l'attaque à la ligne de $LineTypeAfter(bwhile, i, x)$. Cela revient au cas du code linéaire (ligne 14 de l'algorithme).
- i est à l'intérieur de la boucle après l'utilisation de x. Dans ce cas, l'attaque a lieu à partir du deuxième tour de boucle. On peut placer l'attaque en fin de boucle (ligne 17 de l'algorithme).

– **Il n'y a pas d'utilisation de la variable x jusqu'à la fin du bloc**. Le type est défini après le bloc while (ligne 18 de l'algorithme).

En conclusion, chacune des attaques sur la ligne i est couverte par l'attaque définie par la fonction $LineTypeAfter(bloc, i, x)$. ■

Synthèse des codes attaqués à considérer

La section précédente a proposé les définitions des deux fonctions $TypeAfter$ (TA) et $LineTypeAfter$ (LTA) qui permettent de diviser le code d'une fonction en zones où les attaques sont couvertes par une attaque injectée aux numéros de lignes définis par la fonction $LineTypeAfter$. Ainsi, pour simuler toutes les attaques possibles, il n'y a besoin que ne considérer que les codes attaqués aux lignes définies par $LineTypeAfter$.

Théorème 3 *L'ensemble réduit de codes attaqués qui couvre toutes les attaques possibles contre la variable x est le code initial $code_f$, plus les codes attaqués où l'attaque est injectée aux numéros*

de lignes définis par la fonction $LineTypeAfter$, c'est-à-dire :

$$code_f \cup \bigcup_{i \in |program|} Attack(code_f, LineTypeAfter(code_f, x, i), x)$$

Preuve: A l'aide de l'ensemble des 6 lemmes précédents, nous avons démontré que toute attaque en ligne j est soit inopérante, soit couverte par l'attaque définie par la fonction $LineTypeAfter$ en ligne $LineTypeAfter(code_f, x, j)$.

L'ensemble des codes à considérer est donc défini par les valeurs renvoyées par LTA plus le code original. ■

4.3.5 Expérimentations

La réduction proposée en section 4.3 a été implémentée dans un prototype codé en Java. Ce prototype prend un fichier C et détecte automatiquement toutes les variables utilisées. Puis, il recherche toutes les opérations de lecture de ces variables et calcule les lignes correspondant à la fonction LTA qui doit recevoir l'attaque. Pour chaque ligne d'attaque, le prototype génère un nouveau code C. Pour l'exemple du listing 4.1 et la variable `x`, `pin_input` et `res`, il créé les fichiers du listing 4.26.

Listing 4.26– Fichiers créés par le prototyle

```
/home/jf/smart_card/pin_input/attack_line_21.c
/home/jf/smart_card/x/attack_line_11.c
/home/jf/smart_card/x/attack_line_12.c
... (et ainsi de suite pour number1, number2...)
```

Dans la suite de cette section, nous montrons l'effet de notre réduction par couverture d'attaques pour le listing 4.1. Il y a trois variables à considérer : `pin_input`, `x` et `res`. Sans la réduction proposée par le théorème 3, le nombre de codes à générer est grossièrement le nombre de variables multiplié par le nombre de lignes de la fonction considérée, c'est-à-dire 6 (`x` peut être attaqué dans `work()` sur 6 lignes) + 2∗5 (`res` et `pin_input` dans `banking()`) = 16 codes à générer.

Réduction pour `pin_input`

Si l'on applique la réduction à la variable `pin_input`, la ligne 22 est taguée avec *read* par la fonction TA. Donc, pour l'injection de l'attaque, seule la ligne 22 est considérée car $TA(i \in [17..21]) = write$, $TA(i \geq 23) = \emptyset$, $TA(i \in [7..14]) = \emptyset$). Ainsi, appliquer la réduction pour `pin_input` n'introduit qu'une attaque en ligne 22.

Réduction pour x

Si l'on applique la réduction pour la variable `x`, les lignes 10 et 11 sont taguées par *rw* par la fonction TA et la ligne 12 est taguée avec *read*. Pour l'injection des attaques, deux lignes sont concernées car $\forall i \in [10..11], LTA(i) = 11$ et pour $i = 12$, $LTA(i) = 12$ (de manière similaire à `pin_input` les autres lignes ont le type $\emptyset$ ou *write*). Ainsi, appliquer la réduction pour `x` introduit deux attaques en ligne 11 et 12.

Réduction pour res

Dans cet exemple, `res` n'est jamais lu et aucune attaque ne peut avoir d'effet sur lui. Il s'agit clairement d'un exemple pédagogique car `res` n'est jamais utilisé. En pratique il est testé, retourné ou envoyé en tant que paramètre dans un appel de fonction. Dans ces cas, `res` est impliqué dans une lecture et donc une possible attaque.

Résultat de la réduction

En conclusion, appliquer notre méthodologie à `pin_input`, `x` et `res` réduit le nombre d'attaque à 3, au lieu de 16 initialement : une attaque contre `pin_input` en ligne 22 ; deux attaques contre `x` en lignes 11 et 12 ; aucune attaque possible contre `res`.

4.4 Conclusion

Dans ce chapitre, nous avons présenté une méthodologie pour vérifier des propriétés de sécurité exprimées en ACSL. Bien qu'il manque la projection des propriétés du chapitre 3 vers le langage ACSL, ce qui, en soit, n'est pas trivial, nous avons montré que des propriétés simples telles que l'appartenance à un intervalle pour une variable donnée, peuvent être vérifiées à l'aide d'un outil d'analyse statique. Dans les cas où la propriété est difficile à vérifier par manque d'information, nous avons proposé une méthode dynamique permettant d'inférer celles-ci à partir de l'exécution du programme. Cette méthode permet de combler efficacement ce manque d'information qui bloquerait l'analyse statique. Les problèmes liées aux variables globales et les appels de fonctions externes sont ainsi résolues.

Ce chapitre a aussi montré comment le modèle d'attaque peut-être exploité en combinaison avec l'outil d'analyse statique. Nous avons précisément montré comment générer des codes sous attaques physiques à l'aide du modèle d'attaque en provoquant la modification de variables. Ces codes représentent les points d'attaque possibles de l'attaquant. Même si le nombre de ces attaques reste polynomial avec la taille du code, le nombre de codes attaqués générés est très grand et nécessite de s'intéresser à la réduction de ce nombre afin de minimiser les vérifications à effectuer. La fin de ce chapitre est ainsi allé plus loin que l'amélioration du nombre de codes sous attaque : il a prouvé l'optimalité du nombre de codes générés, même pour des codes complexes comprenant des structures de contrôle typiques pour un code C. Un nombre minimum de points d'attaque a donc été trouvé permettant de maximiser l'effet de celles-ci. Les positions ainsi trouvées peuvent être utilisées pour minimiser le nombre d'attaques nécessaires permettant de couvrir l'ensemble des attaques par valeurs possibles. Ainsi, une stratégie de test de sécurité peut être établie.

Enfin, les difficultés relatives à l'utilisation d'un outil d'analyse statique demeurent. Les pointeurs, les tableaux, peuvent rendre la vérification de propriétés difficile. Ces difficultés s'éloignent du cadre de cette thèse, mais les contributions présentées sont indépendantes de ces difficultés. Ainsi, la méthode proposée peut être utilisée avec d'autres outils d'analyse statique ou tirer partie des prochaines avancées dans ce domaine.

Un autre point important est de réussir à travailler sur le code réel de la carte à puce avec de tels outils. Pour réussir à analyser de tels codes, il faut réussir à adapter le code pour le rendre "compatible" avec l'analyseur statique. Ce processus, complètement manuel, peut prendre plu-

sieurs semaines en fonction de la taille du code initial, de ses spécificités et de la chaine de compilation utilisée. Ce processus doit être réitéré si le code source évolue de manière conséquente. Cependant, l'investissement devient rentable car une fois la propriété de sécurité voulue vérifiée, on peut alors exhaustivement tester des attaques et vérifier si la propriété reste valide. Si une grande partie du code est réutilisée de projets à projets, en portant une base commune sur les différentes nouvelles architectures de composant, seul l'effort initial est important. Cependant, le réel investissement se trouve dans la définition, l'écriture et le maintien des annotations exprimant les garanties souhaitées sur le code source. Annoter l'intégralité du code source d'un projet de 100 000 LOC n'est pas envisageable sans des ressources dédiées à cette tâche. Cependant, étant donné que l'intégralité du code source ne remplit pas les mêmes objectifs en terme de sécurité, ces annotations peuvent être mises en place uniquement dans les parties de code sensibles ce qui mitige l'effort à fournir.

4.5 Perspectives

Génération automatique de cas de tests pertinents

La méthode proposée ici consiste à déléguer la détermination de ces informations à une méthode dynamique. Les scénarii intéressants doivent cependant être déterminées manuellement. Des méthodes d'analyse statiques basées sur la sémantique et l'interprétation abstraite permettent cependant d'élaborer des cas de test permettant de stimuler une portion spécifique de code dans un projet [Nori *et al.* 2009]. Intégrer une telle méthode permettrait de réduire la recherche des vecteurs d'entrés nécessaires. La méthode devrait cependant être modifiée afin d'être appliquée dans un contexte de sécurité. Les scénarii qui avantagent l'attaquant devront être distingués de ceux qui ne l'avantagent pas.

Génération automatique de substituts

Des portions de code en assembleur peuvent aussi freiner un analyseur statique. Créer des substituts en C pour ces portions de code et les utiliser lors d'une analyse peut augmenter l'efficacité et la précision de celles-ci. Réussir à créer efficacement ces substituts est une perspective intéressante. L'approche utilisée par [Cifuentes & Gough 1995] pourrait servir de base à la création de ces substituts.

Diminution du champ d'évaluation

Un facteur d'échelle intervient aussi lors de l'analyse de programme de grande taille. Réussir à diminuer le champ considéré lors d'une analyse est un facteur clé pour assurer la vérification de sécurité de l'ensemble d'un système. L'approche adoptée par [Monate & Signoles 2008] pourrait servir de base à une telle réduction.

Identification des points vulnérables du système

Une contrainte consiste aussi à isoler les parties du programme qui représentent un danger d'un point de vue fonctionnel si elles sont soumises à des attaques physiques. Dans le cha-

pitre 5 propose une méthode permettant d'identifier ces points de vulnérabilité à l'échelle d'un programme.

Chapitre 5

Test de sécurité et caractérisation visuelle de programme sous attaques physiques simulées

Sommaire

5.1 Introduction . . . 141
5.2 Méthodologie de test proposée . . . 143
5.2.1 Principe . . . 143
5.2.2 Méthode de classification des résultats . . . 146
5.3 Expérimentations et analyse sécuritaire . . . 148
5.3.1 Plate-forme expérimentale . . . 149
5.3.2 Exemple sur BZIP2 . . . 150
5.3.3 Exemple sur GZIP . . . 153
5.3.4 Couverture assembleur par le modèle C . . . 157
5.4 Analyse des résultats . . . 160
5.4.1 Distribution des attaques par taille du saut . . . 160
5.4.2 Analyse visuelle des fonctions vulnérables . . . 161
5.4.3 Implémentation de contre-mesures . . . 163
5.5 Résultats expérimentaux sur code de carte à puce . . . 164
5.6 Conclusion . . . 166
5.7 Perspectives . . . 167

5.1 Introduction

Dans ce chapitre, nous introduisons une contribution méthodologique permettant de tester si les sécurités mises en place dans un code source sont capables de résister à des attaques physiques. Nous donnons les résultats expérimentaux de cette méthode déployée à l'échelle d'un projet de carte à puce d'Oberthur Technologies.

Une approche possible pour déterminer si une attaque réussit consiste à énumérer exhaustivement toutes les attaques possibles, puis d'analyser l'impact de chacune d'elles sur l'exécution du programme, afin de pouvoir discerner celles qui ont réussi des autres. Étant donné que les conséquences des attaques physiques peuvent être expliquées au niveau assembleur, une telle approche doit a priori être effectuée à ce niveau. Ainsi, en étudiant les conséquences des attaques au niveau assembleur, on peut établir un modèle d'attaque à ce niveau (par une modification ou insertion de code simulant l'effet de l'attaque). On peut ainsi découvrir de nouvelles attaques

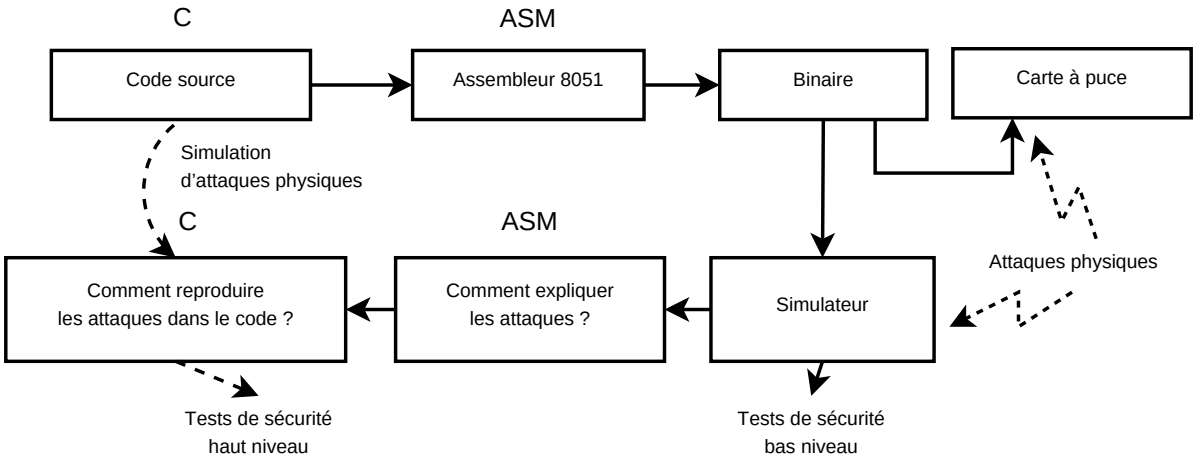


FIGURE 5.1 – Schéma de la méthodologie adoptée pour l'analyse sécuritaire

susceptibles de réussir : soit en exécutant la version attaquée du code et en comparant le comportement obtenu au comportement attendu dans un scénario sans attaque, soit en analysant statiquement la version attaquée du code et en prouvant des propriétés de sécurité. Cependant avec cette méthode, une fois qu'une attaque a été identifiée comme potentiellement dangereuse, ses conséquences sur le code assembleur doivent encore être reliées au code source original au niveau C, afin d'implémenter une contre-mesure. Cette tâche est particulièrement difficile. En effet, il n'est pas évident de décompiler du code assembleur afin de retrouver le code source original. De plus, la réussite de cette décompilation dépend en grande partie du compilateur et du niveau d'optimisation utilisé au moment de la génération du binaire.

C'est donc pourquoi, en utilisant le modèle d'attaque à haut niveau présenté en chapitre 3, nous proposons d'exploiter le modèle de haut niveau afin d'étudier l'impact de chaque attaque sur l'exécution du programme. Le fait d'utiliser un modèle d'attaque à haut niveau est symbolisé dans la figure 5.1 par la flèche de gauche en pointillés. Comparé à un cycle qui incorpore des attaques simulées ou encore des attaques sur cible matérielle, cette méthode permet d'obtenir plus rapidement des résultats en raccourcissant la prise en compte des attaques à l'intérieur du cycle de développement. Le développeur peut ainsi savoir au niveau du code source qu'il développe si des attaques physiques sont susceptibles de compromettre son implémentation.

Nous utilisons donc le modèle d'attaque établi en chapitre 3 afin de réaliser la simulation des attaques dans le code source C. Nous nous intéressons particulièrement aux attaques affectant le contrôle de flot du programme. Dans la section 5.2, nous présentons la méthodologie proposée qui utilise des simulations d'attaques sur le contrôle de flot au niveau C afin d'identifier des attaques dangereuses qui sont susceptibles de compromettre la sécurité du programme. Cette méthodologie permet au développeur de faire une analyse poussée de l'ensemble des impacts possibles d'attaques physiques son code. Nous présentons les résultats expérimentaux d'une telle analyse en section 5.3. Aucune expérimentation n'est donc nécessaire au niveau assembleur. Nous nous servons des programmes de compression BZIP2 et GZIP, dont le comportement symbolise celui d'une carte à puce, pour montrer comment la méthodologie proposée permet de caractériser des programmes soumis à des attaques physiques. Afin de montrer le rendement de la méthode proposée, i.e., la couverture des attaques possibles à bas niveau par celles simulées à haut niveau,

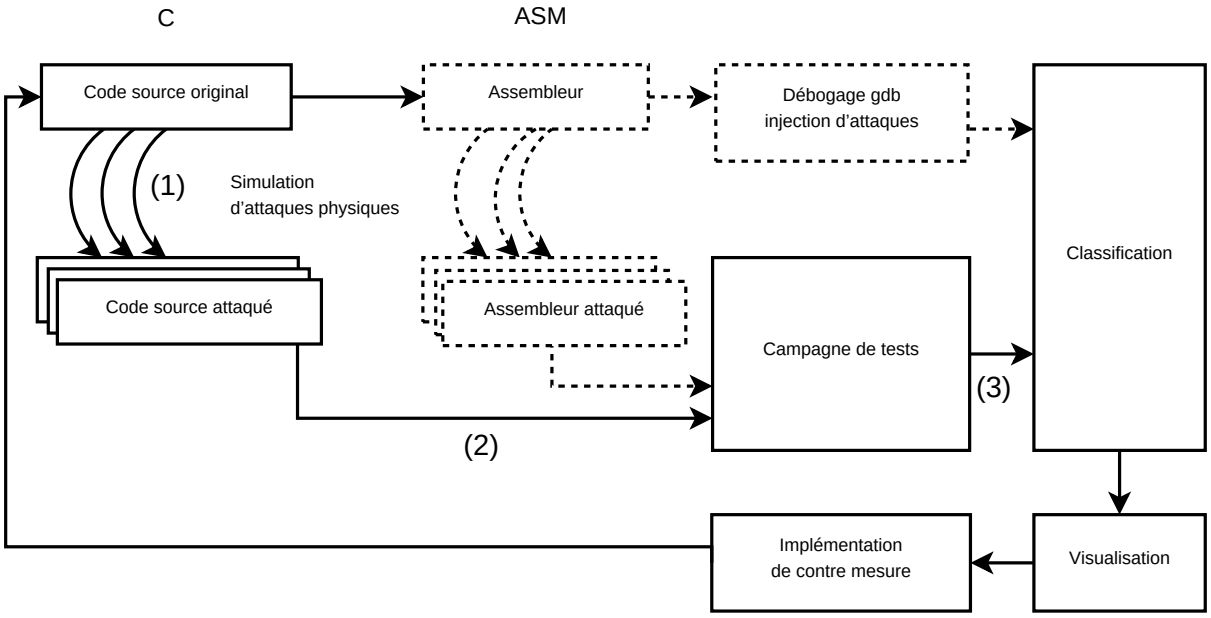


FIGURE 5.2 – Vue d'ensemble de la plate-forme expérimentale

une sous section 5.3.4 est dédiée à la comparaison des comportements du système obtenus suite à injection d'attaques sur le contrôle de flot au niveau C avec ceux obtenus au niveau assembleur. La section 5.4 analyse les résultats obtenus grâce à la méthode proposée et montre comment ces résultats peuvent être utilisés par le développeur pour améliorer la sécurité de son code. Cette section montre également comment la méthodologie permet d'évaluer l'efficacité d'une contre-mesure. Finalement, la section 5.5 conclut en proposant des résultats concrets obtenus sur le code d'une vrai carte à puce. Les contributions couvertes dans ce chapitre font référence à la publication [Berthomé *et al.* 2012].

5.2 Méthodologie de test proposée

Le chapitre 3 a identifié comment il est possible de modéliser les attaques sur le contrôle de flot au niveau du code source. Cette section décrit la méthodologie de test que nous proposons représentée dans la figure 5.2 par des flèches pleines. Les flèches en pointillé de la figure représentent la validation expérimentale de cette proposition.

5.2.1 Principe

Le principe de la méthodologie proposée, consistant à attaquer fonctionnellement un programme de manière exhaustive, est proche de celle présentée dans [Srivatanakul *et al.* 2005]. Ce principe, utilisé notamment en sécurité dans les tests d'intrusion, vise à tester les systèmes en les soumettant à des attaques variées. La présence d'une faille est supposée dans le système, le but est de la mettre en évidence par ces attaques. Cette méthodologie cible l'implémentation de fonctionnalités mais aussi les spécifications de celles-ci, permettant de chercher des failles aussi bien dans l'une que dans les autres. Concrètement, la méthode employée consiste à stimuler des

points d'attaques et à observer la réaction du système. Cette technique est notamment utilisée dans le domaine de l'injection de faute par l'ajout de *saboteurs* qui perturbent le système lors de son exécution [Grinschgl *et al.* 2011].

Notre méthodologie de test propose d'utiliser les attaques par saut du modèle d'attaque par GOTO afin de tester exhaustivement le code source d'un scénario fonctionnel soumis à l'ensemble des attaques de contrôle de flot possibles. Un scénario fonctionnel est une suite de commandes envoyées à la carte afin de déclencher des actions au sein de celle-ci. Ces actions ont pour but de remplir une ou plusieurs tâches qui sont les objectifs fonctionnels du scénario. Généralement, pour chaque fonctionnalité décrite dans les spécifications fonctionnelles du produit, il existe un scénario fonctionnel permettant de stimuler cette fonctionnalité implémentée au travers d'un test fonctionnel. Pour un *Verify Pin*, il existe déjà des tests fonctionnels qui, par exemple, simulent un scénario où l'utilisateur présente un PIN correct (respectivement, incorrect) et vérifient que le jeton d'authentification renvoyé est correct (respectivement incorrect). Des variantes de ces scénarii existent dans les tests unitaires, les campagnes de tests fonctionnels ou les tests de certification permettant d'évaluer la correspondance de l'implémentation aux spécifications fonctionnelles. Les développeurs responsables de l'implémentation auront généralement créé de tels tests. La technique d'injection d'attaque dans le code source du programme étant totalement indépendante de la technique utilisée pour effectuer les tests, elle s'applique donc quelle que soit la méthodologie ou le niveau de test utilisé. Tests unitaires, tests d'intégration ou tests de régression retourneront des informations d'un niveau différent mais incorporant l'effet d'attaques physiques. Ainsi, un des avantages de cette méthode est qu'elle permet de réutiliser ces tests fonctionnels en les dérivant pour créer des tests de sécurité. Pour cela, il suffit de modifier le test pour faire apparaître un avantage possible pour l'attaquant vis-à-vis du test en question. Par exemple, pour le *Verify PIN*, modifier le PIN présenté par l'utilisateur par un PIN incorrect. Si le test ainsi modifié retourne le même résultat que le test non modifié, l'attaquant a trouvé un moyen par une attaque physique d'entrer un PIN incorrect mais d'obtenir un jeton d'authentification correct. On peut remarquer que si les tests étaient effectués sans changer les scénarii, on testerait avec cette méthode la résilience d'une exécution par rapport à des attaques.

Une fois la campagne de test d'attaques exhaustives effectuée, nous proposons de classifier les résultats obtenus afin de différentier les attaques qui ont réussi des autres. Les auteurs de [Madeira *et al.* 2000] adoptent une classification similaire à celle présentée ici mais dans un contexte sans sécurité et appliquée à la découverte d'erreurs dans un programme. Les auteurs catégorisent les comportements de 3 programmes écrits en C face à des injections de fautes émulées et créent un parallèle avec l'origine logicielle de la faute. Les résultats montrent que les outils d'injection de fautes par émulation réussissent à simuler certaines erreurs à haut niveau comme les erreurs d'affectation, de logique, d'interface ou de partage de ressources. Cependant, certaines d'entres elles comme les erreurs qualifiées d'algorithmiques ou de fonctionnelles se prêtent mal à des simulation de bas niveau. Plus complexes, car nécessitant une modification extensive de haut niveau pour être apparentes, celles-ci nécessiteraient une intervention manuelle pour définir et déclencher précisément l'erreur souhaitée. Les auteurs citent [Christmansson & Chillarege 1996] qui mentionne que ces erreurs non couvertes représentent 44 % des erreurs logicielles trouvées lors de leurs expériences. Ces chiffres renforcent l'intérêt que représentent une solution logicielle pour la découverte de nouvelles fautes dans l'implémentation. Les auteurs de [Madeira *et al.* 2000] présentent également que des métriques peuvent être utilisées comme moyen de guider l'injection des fautes pour les rendre plus pertinentes et efficaces. Ces métriques permettent de compenser

le manque de résultats "terrains" qui peuvent normalement aider à guider ces injections. Nous présenterons, dans la section 5.2.2, une utilisation différente de telles métriques non pas comme guide mais comme un moyen de caractériser une attaque en synthétisant les formes de code présentes dans le code sauté par l'attaque.

Dans la littérature, différentes autres approches ont été adoptées afin de vérifier la sécurité des codes embarqués contre les attaques physiques. Ces approches par modélisation ou simulation font intervenir des niveaux d'abstraction différents ainsi que des méthodes de vérification différentes.

- Une modélisation en SystemC d'une partie du système au niveau matériel et l'introduction d'attaques dans ces modèles [Rothbart *et al.* 2004] ;
- La création d'une plate-forme d'injection de fautes pour les cartes s'inspirant des plates-formes d'injection existant en sureté de fonctionnement et permettant l'insertion d'attaques par délai en prenant l'AES pour exemple [Faurax 2008] ;
- Des simulateurs de composants dédiés à la modélisation des fuites d'informations sur les algorithmes cryptographiques [Andouard 2009] [Agostini 2009] ;
- Une plate-forme de création de codes mutants en mémoire pour la vérification de propriétés de sécurité sur des applications Java Card [Machemie *et al.* 2011].

Malheureusement, aucune de ces méthodes ne porte sur un système C complet dans le cas d'attaques par fautes. S'il est possible que certaines de ces méthodes passent à l'échelle, elles ne permettent pas d'obtenir un résultat direct sur le code attaqué mais seulement sur un modèle de celui-ci à différents niveaux d'abstraction. Le développeur qui doit introduire une contre-mesure dans le code pour corriger une vulnérabilité trouvée doit transcrire les résultats obtenus à ce niveau d'abstraction au niveau du code source, ce qui demande un effort supplémentaire. Finalement, une modélisation au niveau du composant, si elle a l'avantage d'être plus fidèle, demande une adaptation de celui-ci dans le cas d'une utilisation avec un composant aux spécificités différentes.

La méthodologie proposée a pour but de passer à l'échelle au niveau d'un projet C complet. Par rapport à [Madeira *et al.* 2000], elle propose une granularité supérieure sur un jeu d'entrée restreint à un scénario particulier intéressant du point de vue de la sécurité. L'objectif n'est pas de mettre en évidence des erreurs logicielles par des tests mais de découvrir quelles attaques logicielles entrant dans un modèle d'attaque peuvent compromettre la sécurité du système. Les résultats présentés font donc apparaître à l'échelle d'une fonction, pour toutes les fonctions d'un programme, les effets d'une campagne exhaustive d'attaques par saut. Les résultats *incorrect* de [Madeira *et al.* 2000] sont ici décomposés en ERROR, où le système se rend compte d'un problème, et BAD, qui peuvent représenter un gain pour l'attaquant. Nous proposons de visualiser dans l'espace ces attaques et les résultats obtenus afin de localiser précisément les zones de code sensibles. Cette méthodologie permet au développeur, grâce à l'identification de ces zones, d'ajouter des contre-mesures appropriées. Ces contre-mesures peuvent être à nouveau testées en utilisant la même méthode afin de s'assurer que celles-ci empêchent la réussite de l'attaque et aussi n'introduisent pas de nouvelle faille exploitable par une autre attaque. Pour des exemples de contre-mesures ainsi qu'une méthode alternative de vérification, le lecteur peut se référer à [Sere *et al.* 2011]. Les auteurs utilisent un interpréteur abstrait, fonctionnant sur un modèle de la mémoire, ainsi qu'un générateur de codes mutants, basé sur un modèle d'attaque, afin d'améliorer la résistance du code contre les attaques par fautes. Leur outil s'utilise sur du bytecode Java Card. Le développeur peut annoter des fonctions dans le code afin que

la machine virtuelle effectue des vérifications supplémentaires d'intégrité en utilisant un jeu de contre-mesures intégré à la machine virtuelle. Certaines techniques utilisées sont proches de celles présentées dans ce chapitre ou le précédent. Le concept d'interprétation abstraite est aussi utilisé par Frama-C. L'approche par génération de codes mutants est proche de l'approche qui consiste à créer de multiples versions attaquées d'un code original que nous avons adoptée ici. La principale différence est que la génération de codes mutants est effectuée directement en mémoire tandis que la solution proposée ici travaille par instrumentation du code source en amont du compilateur. Cette solution offre une meilleure généricité en C qui ne peut pas s'appuyer sur une machine virtuelle. Dans ce chapitre, nous proposons également une classification des effets des attaques générées au niveau fonctionnel afin de faciliter le travail d'identification du code vulnérable.

5.2.2 Méthode de classification des résultats

Quand un programme contenant une attaque simulée (par injection au niveau assembleur, C ou avec GDB) est exécuté, le comportement du programme doit être évalué afin de déterminer si l'attaque a réussi. Nous proposons la classification suivante qui regroupe les différents cas d'exécutions :

- NOT TRIGGERED : l'attaque n'a pas été déclenchée lors de l'exécution. Ce cas apparaît lorsqu'une attaque est insérée dans une portion de code non stimulée par le scénario d'exécution choisi. Dans tous les autres cas, l'attaque est déclenchée ;
- SIGNAL : le programme se termine et retourne un signal (SIGSEGV, SIGBUS, ...) ;
- ERROR : le programme se termine et retourne un message d'erreur ;
- GOOD : le programme se termine et la sortie retournée ainsi que le comportement du programme correspondent à ceux attendus ;
- BAD : le programme se termine mais la sortie retournée ainsi que le comportement du programme ne correspondent pas à ceux attendus. La définition de BAD doit être déterminée pour chaque application étant donné que le comportement ou la sortie attendue dépendent de l'application considérée ;
- KILL : le programme ne se termine pas et doit être interrompu.

Les exécutions aboutissant à des GOOD ou ERROR peuvent être considérées comme tolérantes vis-à-vis de l'attaque : le programme est capable de faire face à l'attaque en l'ignorant ou en détectant un comportement incorrect. Pour une carte à puce, détecter une erreur peut amener à "tuer" la carte (en effacer le contenu de sa mémoire) ce qui est considéré comme un comportement approprié d'un point de vue sécuritaire. Un CRASH, un SIGNAL ou un KILL est aussi un comportement approprié pour l'expert en sécurité. Cela veut dire que le programme a été mis dans un état instable qui amène au blocage ou la non terminaison de celui-ci. Par conséquent, un attaquant n'est pas en mesure d'exploiter une telle attaque étant donné que la transaction effectuée avec le terminal n'aboutit pas. Par rapport à [Madeira *et al.* 2000], les correspondances sont *correct* pour GOOD, *hang* pour KILLED et *crash* pour SIGNAL.

Le dernier scénario, le plus dangereux d'un point de vue de la sécurité, est une exécution BAD. Ce scénario peut être interprété comme une exécution qui a abouti mais a été suffisamment perturbée pour induire un comportement incorrect. Ces scénarii sont les plus importants pour le développeur et nécessitent que une investigation plus approfondie de sa part afin d'implémenter la contre-mesure adéquate.

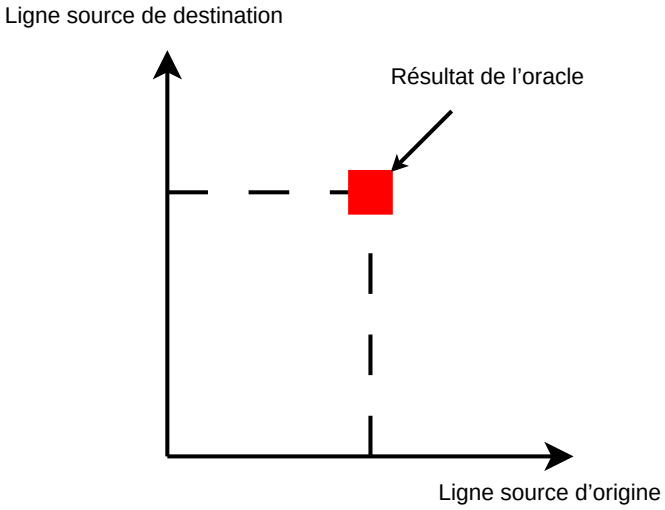


FIGURE 5.3 – Représentation 2D des résultats donné par un oracle

Nous qualifions comme *oracle* le processus permettant de déterminer si le comportement du programme est correct ou incorrect par rapport à un scénario donné. La notion d'oracle est développée d'un point de vue pratique dans la section 5.5. La figure 5.3 illustre la représentation en deux dimensions que nous avons choisie pour classifier les résultats des attaques injectées suivant le modèle d'attaque par GOTO. L'axe des abscisses donne la ligne de code source de départ du correspondant à un `goto` injecté dans le code à une ligne précise. L'axe des ordonnées donne la ligne de code source d'arrivée du saut correspondant à un `label` injecté dans le code. L'intersection de ceux deux axes est un carré de couleur qui correspond à une des classifications présentées précédemment qui traduit un comportement particulier du système. Chacune des classes possède un code couleur spécifique. Cette représentation en deux dimensions permet avec l'aide des codes couleurs de regrouper et d'identifier visuellement les zones de code qui, si elles ne sont pas exécutées du fait d'une attaque par saut, ont un comportement similaire.

Synthèse pour une attaque

Finalement, quand une attaque est identifiée, une quantification automatique des caractéristiques du code sauté est réalisée. Cette quantification aide à déterminer la nature de l'attaque réalisée. Par exemple, le listing 5.1 montre une synthèse des attaques classées en tant que BAD qui saute l'extrait de code suivant, de la ligne 4116 à la ligne 4133 de la fonction BZ2_blockSort :

La quantification recense le nombre d'affectations et d'appels de fonction qui n'ont pas été réalisés. Elle prend aussi en compte si l'attaque entre ou sort d'un bloc de code d'une profondeur supérieure. Dans cet exemple, les variables affectées ne sont pas utilisées dans le reste de la fonction, ce qui veut sans doute dire que ne pas réaliser l'appel à `mainSort` entraîne une réussite de l'attaque. Étant donné que le développeur sait ce que doit réaliser son code, il doit être en mesure de facilement comprendre les conséquences du saut de ces lignes de code C. Ainsi, la vue d'ensemble des éléments de code permet à celui-ci d'appréhender rapidement la cause et les effets précis de cette attaque et par conséquent, l'aide à implémenter la contre-mesure la plus appropriée. Celle-ci peut par exemple assurer que la fonction `mainSort` est bien appelée.

Listing 5.1– Exemple d'une zone de code sautée

```
goto label;                 // début de zone d'attaque
if (wfact > 100) {
    wfact = 100;
}

budgetInit = nblock * ((wfact - 1) / 3);
budget = budgetInit;

mainSort(ptr, block, quadrant, ftab, nblock, verb, &budget);

if (verb >= 3)
    VPrintf3("      %d work, %d block, ratio %5.2f\n",
             budgetInit - budget,
             nblock,
             (float)(budgetInit - budget) /
             (float)(nblock == 0 ? 1 : nblock));

if (budget < 0) { label:    // fin de zone d'attaque
```

Délai d'expiration

Lors des expérimentations, en plus d'une synthèse des éléments sautés lors de l'attaque, nous avons dû prendre en compte un élément supplémentaire qui est le délai d'expiration maximal après lequel un test doit être interrompu. Cette interruption est nécessaire dans les cas où une attaque provoque une boucle infinie dans le programme. Un paramètre expérimental, le délai avant interruption du test ou *délai d'expiration* a donc été établi. Nous avons vérifié expérimentalement qu'un facteur de deux fois le temps d'exécution maximal pour le délai d'expiration convenait. La figure 5.4 montre la classification de toutes les attaques possibles contre la fonction principale de BZIP2 en variant le délai d'expiration pour la méthodologie utilisant GDB. En utilisant GDB, il est possible de mesurer le temps d'exécution de la fonction principale avant l'interruption du processus si l'exécution n'a pas atteint la fin de la fonction. En utilisant un délai d'expiration de 0.1 s, un grand nombre d'exécutions sous attaques sont classées en tant que KILLED : la compression n'a pas le temps nécessaire pour être effectuée. La figure 5.4 montre qu'avec un délai de 0.14 s, la classification des attaques se stabilise. Étant donné que le temps d'exécution original de BZIP2 est approximativement de 0.159 s, cela montre qu'un facteur de 2 avant l'interruption du processus est suffisant pour avoir des résultats représentatifs.

5.3 Expérimentations et analyse sécuritaire

Une plate-forme expérimentale a été développée afin de remplir trois objectifs.

- Premièrement, confirmer que le modèle d'attaque au niveau C est cohérent avec le modèle d'attaque au niveau assembleur (ce modèle d'attaque au niveau assembleur correspond au modèle de faute sur un octet que nous avons choisi dans le chapitre 3) ;
- Deuxièmement, montrer comment le modèle d'attaque au niveau C permet au développeur d'obtenir des informations pertinentes sur les failles de sécurité dans son code ;
- Troisièmement, montrer que l'efficacité d'une contre-mesure peut être évaluée visuellement.

La première section 5.3 présente l'installation de la plate-forme expérimentale dans une première sous section 5.3.1. Ensuite, les sections 5.3.2 et 5.3.3 présentent les taux de couvertures

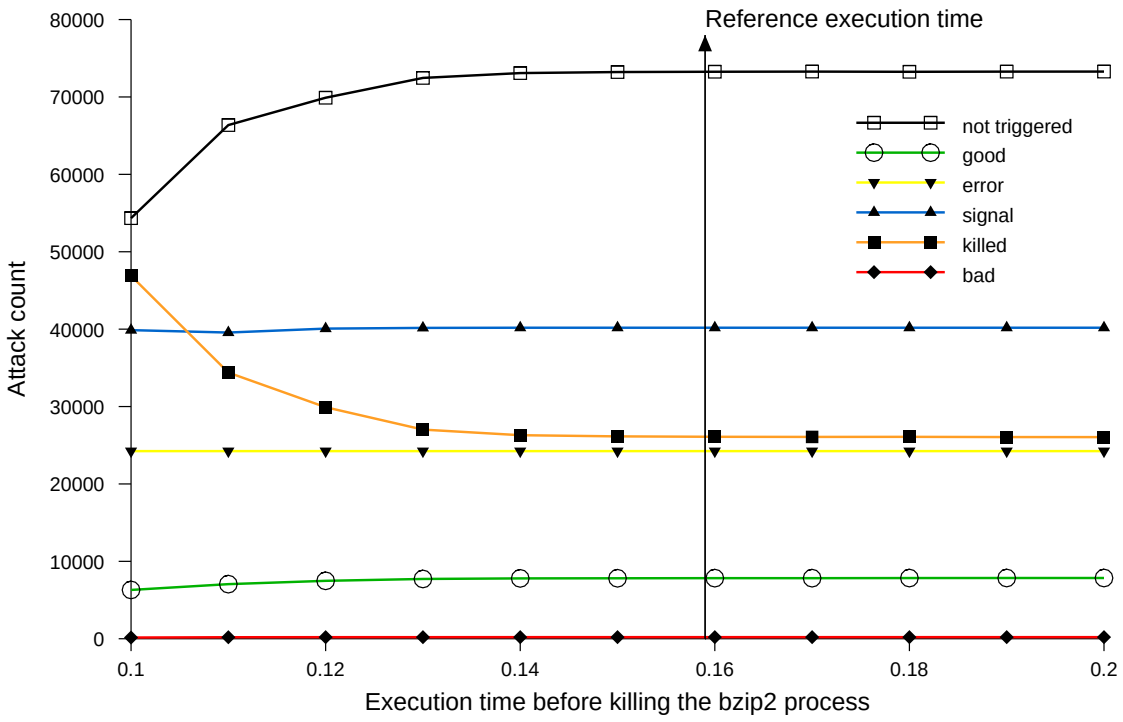


FIGURE 5.4 – Nombre d'attaques pour chaque catégorie en fonction du délai d'expiration (en utilisant la méthode GDB)

entre les différentes méthodes employées pour injecter des fautes dans BZIP2 et GZIP et montre comment évaluer l'efficacité d'une contre-mesure. Une troisième sous section 5.3.4 présente une comparaison entre ces différents taux de couverture. La seconde section 5.4 propose une méthode d'analyse des résultats basée sur la différentiation fonctionnelle des attaques efficaces et de celles qui ne le sont pas. Les critères permettant cette différentiation ont été exposés dans la section 5.2.2. Cette méthode d'analyse permet à la fois l'identification de vulnérabilités dans le code mais aussi l'évaluation d'une contre-mesure implémentée. Finalement, la 5.5 montre comment la méthodologie proposée peut aider dans l'analyse de la sécurité d'un extrait de code de carte à puce d'Oberthur Technologies.

5.3.1 Plate-forme expérimentale

Comme expliqué précédemment, nous considérons uniquement les attaques impactant le contrôle de flot causé par des attaques par NOP ou des attaques par saut. Nous avons par conséquent basé l'implémentation de notre méthodologie sur le modèle d'attaque par GOTO afin d'injecter exhaustivement ces attaques par GOTO au niveau du code source C.

Les binaires obtenus sont lancés et le résultat de chaque exécution est classifié suivant la méthodologie décrite dans la section 5.2.2. L'injection peut être effectuée dans le code source avant compilation ou dynamiquement pendant l'exécution en utilisant un débogueur. Afin de

pouvoir analyser les taux de couverture, nous avons aussi implémenté l'injection de ces attaques par saut au niveau assembleur. Trois méthodes d'injection ont donc été utilisées : au niveau du code source C, assembleur et en utilisant un débogueur :

- La méthodologie d'*injection d'attaque* GOTO *assembleur* insère des sauts inconditionnels au niveau assembleur : une instruction de saut inconditionnel JMP LABEL ainsi qu'un label de destination LABEL sont insérés dans le corps de la fonction étudiée. Le code résultant est alors compilé afin d'obtenir une version attaquée du programme ;
- La méthodologie d'*injection d'attaque* GOTO *C* insère directement une attaque par GOTO dans le code source de la fonction ciblée. Le code instrumenté est ensuite compilé et exécuté. Comme montré dans la section 3.4.3 du chapitre 3, les attaques transientes peuvent être simulées au niveau C en encapsulant les attaques par GOTO injectées dans une condition `if/then` qui contrôle lorsqu'elle est déclenchée en utilisant un compteur d'exécution. Nous avons aussi implémenté l'injection d'attaques transientes au niveau C ;
- La méthodologie d'*injection d'attaque* GOTO *utilisant* GDB évite d'avoir à compiler une nouvelle fois le programme : le débogueur GDB est utilisé avec le binaire d'origine et interrompt à la volée l'exécution du programme afin d'injecter dynamiquement au niveau d'une ligne C une attaque GOTO permettant de faire sauter l'exécution à une autre endroit du programme. En utilisant les interfaces de programmation python de GDB, un script contrôle le programme s'exécutant sous GDB. Un point d'arrêt est inséré au début de la fonction étudiée. Quand l'exécution atteint ce point d'arrêt, une duplication du processus en cours est réalisée. Le script python insère ensuite un point d'arrêt à la ligne source de l'attaque par GOTO. Quand la source du saut est atteinte, un saut est réalisé en précisant à GDB le numéro de la ligne cible qui convertit celle-ci en une adresse mémoire. L'exécution du programme est ensuite reprise jusqu'à son terme. L'avantage principal de cette méthode est qu'une fois qu'une attaque a été réalisée, l'attaque suivante peut être testée en utilisant une nouvelle duplication du processus original (qui a été interrompu au début de la fonction étudiée).

Des attaques par saut peuvent modifier arbitrairement le contrôle de flot permettant de sauter du corps d'une fonction dans le corps d'une autre fonction. Cependant, un tel saut a une faible probabilité de succès étant donné que la pile et donc le contexte d'exécution deviennent rapidement incohérentes une fois que les paramètres sont dépilés. Par conséquent, nous considérons uniquement les attaques par saut intra-fontion.

5.3.2 Exemple sur BZIP2

Nous considérons le programme de compression BZIP2 inclus à l'intérieur des études comparatives de référence SPEC 2006 [Henning 2006] afin d'effectuer une analyse de sécurité en profondeur. L'application est constituée de 106 fonctions et 3 069 instructions C. Le code assembleur correspondant est composé de 26 006 instructions. Le principe de fonctionnement de BZIP2 est basé sur l'algorithme de Burrows-Wheeler et un encodage de Huffman. L'algorithme de Burrows-Wheeler repose sur une réorganisation des données à compresser de manière à regrouper les caractères identiques à l'aide d'opérations de décalage. L'encodage de Huffman repose sur un encodage probabiliste des données d'entrée en un arbre binaire trié à l'aide des fréquences d'occurrence de chaque caractère. Le tri est effectué à l'aide d'opérations de décalage et d'additions.

Nous sélectionnons comme entrée un fichier de 489K contenant des informations aléatoires. Afin d'analyser la sortie d'une compression, nous considérons une exécution comme mauvaise (BAD) toute exécution qui génère un fichier compressé différent de celui de référence obtenu avec une exécution sans attaque du programme.

La figure 5.5 montre la répartition des résultats obtenus par l'oracle classifiés suivant les catégories mentionnées dans la section 5.2.2. Ce graphique montre non seulement la trace d'exécution du scénario de test utilisé à l'intérieur de BZIP2, il montre aussi, pour cette trace, les fonctions qui, si elles sont attaquées, peuvent avantager un attaquant. Ainsi, les parties blanches (NOT TRIGGERED) du graphique correspondent au code qui n'est pas stimulé par le test utilisé. Une grande proportion de vert (GOOD) pour une fonction signifie qu'elle est résistante aux attaques par fautes. Une grande proportion de jaune (ERROR) signifie que le code dispose de mécanismes de détection qui sont déclenchés par l'attaque. Une grande proportion de bleu (SIGNAL) montre que le programme communique avec le système d'exploitation par le biais de signaux pour signaler un comportement anormal. Ce cas ne concerne pas les systèmes embarqués qui n'utilisent pas de signaux. Une grande proportion d'orange pour une fonction montre qu'en déclenchant une attaque dans cette fonction une boucle infinie a été créée et que le test a été arrêté après un délai calculé dans la section 5.2.2. Généralement, ce cas survient lorsque l'attaque introduit une boucle ou empêche l'exécution de la condition d'arrêt d'une boucle existante. Finalement, les fonctions qui ont une grande proportion de rouge (BAD) sont particulièrement vulnérables aux attaques par fautes. Attaquer de telles fonctions augmente la probabilité d'obtenir une sortie erronée sans que le programme ne le détecte. Grâce à cette représentation, il est possible de caractériser visuellement la résistance aux attaques du programme fonction par fonction et ainsi discriminer parmi ces fonctions, celles qu'il convient de sécuriser.

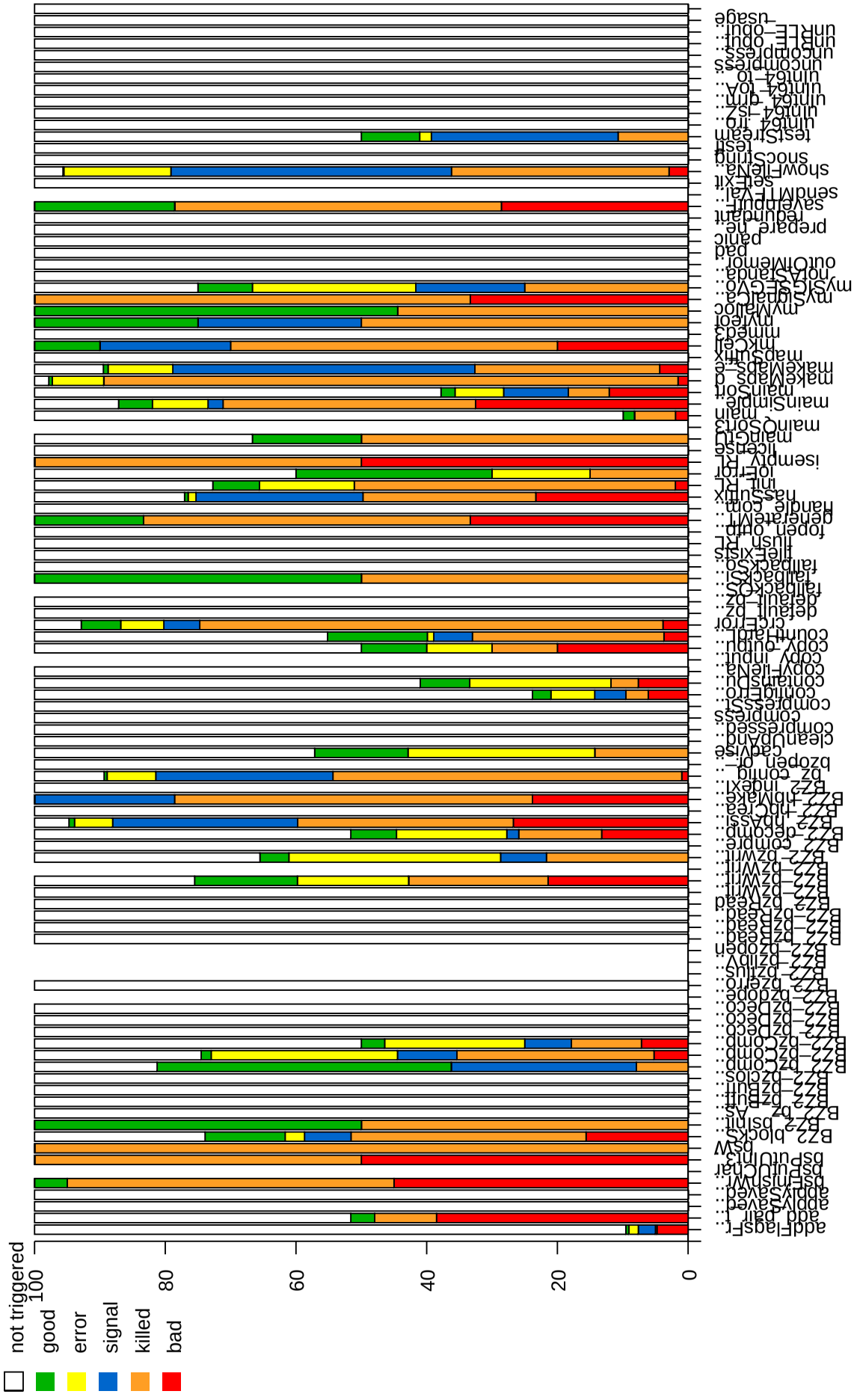


FIGURE 5.5 – Résultats d'attaques permanentes réalisées en C sur le code de BZIP2

5.3.3 Exemple sur GZIP

Nous avons également utilisé la méthodologie sur les fonctions du code de GZIP afin de pouvoir comparer le comportement de deux programmes sous attaques. Ces deux programmes sont deux implémentations différentes d'un même besoin fonctionnel et reposent sur des algorithmes différents. Nous considérons maintenant le programme de compression GZIP. Cette application est constituée de 61 fonctions et 2 453 instructions C. Le code assembleur correspondant est composé de 12 951 instructions.

Le principe de fonctionnement de GZIP est basé sur un encodage de Lempel-Ziv et un encodage de Huffman. L'encodage de Lempel-Ziv repose sur la création d'une table de symbole créée en parcourant l'entrée à compresser. Celle-ci est basée sur des distances calculées entre un motif courant et un motif présent dans une partie précédente du fichier. L'implémentation d'un tel encodage nécessite de parcourir les données et donc des manipulations d'index et des calculs de distance.

Nous sélectionnons comme entrée le même fichier de 489K contenant des informations aléatoires ayant servi aux tests sur BZIP2. Afin d'analyser la sortie d'une compression, nous considérons une exécution comme mauvaise (BAD) toute exécution qui génère un fichier compressé différent de celui de référence obtenu avec une exécution sans attaque du programme.

Ainsi, la figure 5.6 montre les résultats obtenus sur l'ensemble du code de GZIP. Comme pour BZIP2, la trace obtenue caractérise une exécution particulière du code par l'utilisation d'une partie du code du programme. Il est possible, à partir de la figure, de déterminer les fonctions sensibles qui auront une plus grande proportion de rouge. Il est aussi possible de comparer à partir de ces "génomes", deux programmes différents exécutant un même scénario afin de déterminer lequel est plus résistant aux attaques par fautes.

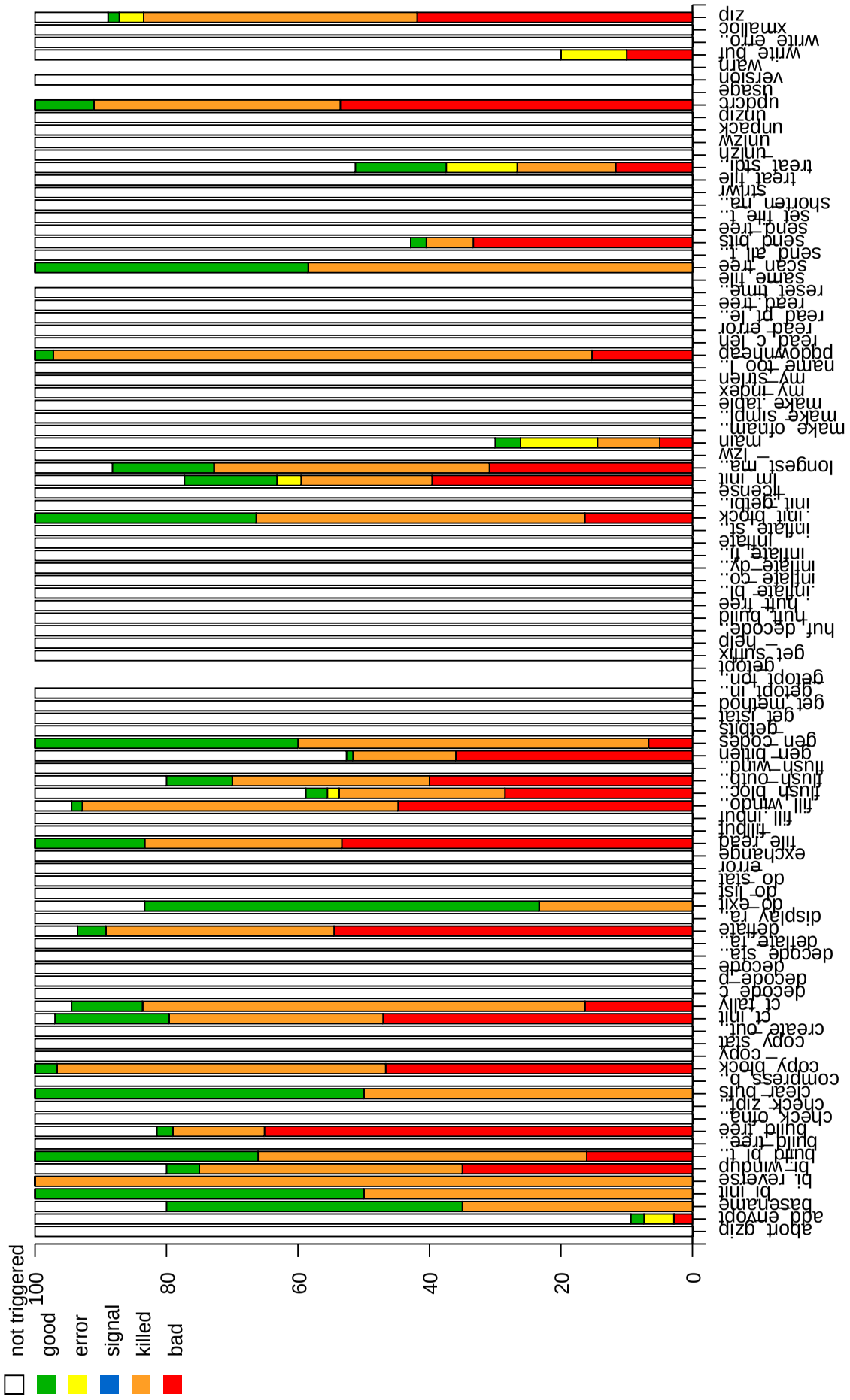


FIGURE 5.6 – Résultats d'attaques permanentes réalisées en C sur le code de GZIP

Comparaison de génomes

BZIP2 produit une sortie plus petite et repose sur la création de blocs indépendants ce qui permet une éventuelle récupération en cas de corruption. GZIP est légèrement moins performant en terme de facteur de compression mais est un peu plus rapide. La consommation mémoire à l'exécution est aussi plus faible pour GZIP [Collin 2005] ce qui laisse penser que le programme utilise moins de variables de grande taille simultanément. En effet, le fonctionnement de GZIP repose sur le principe d'une fenêtre de taille fixe parcourant le fichier source en analysant une portion limitée du fichier à compresser tandis que pour BZIP2 l'ensemble des permutations de la chaine d'entrée doit être contenue à un moment donné en mémoire.

Nous qualifions de "génome" une représentation de la répartition des classes de résultats obtenues après l'exécution d'un scénario de test en présence d'attaques. En effet, cette représentation caractérise le comportement du programme sous attaques et crée une signature particulière de celui-ci dans un cas particulier d'exécution s'il est soumis à des attaques par saut.

Les figures 5.7 et 5.8 illustrent les "génomes" des deux programmes de compression. Les fonctions et le code qui n'ont pas été stimulés ont été retirés afin d'obtenir des échantillons représentatifs du comportement de chaque programme. Les différences entre les deux programmes commencent à devenir visibles. On remarque un plus grand nombre d'erreurs ERROR et de SIGNAL dans le génome de BZIP2 que dans celui de GZIP. Cette observation nous montre que BZIP2 embarque plus de code dédié à la gestion des erreurs et utilise plus de signaux dans l'implémentation de ses fonctionnalités.

Afin de pouvoir comparer plus efficacement ces deux programmes, la figure 5.9 illustre la comparaison entre les génomes, en distribuant les attaques par classes. On peut ainsi confirmer les observations précédentes et confirmer que GZIP est plus sensible à des attaques par fautes que BZIP2. Le plus grand nombre de mécanismes d'erreur de BZIP2 visibles dans les catégories SIGNAL et ERROR sont vraisemblablement responsables de cette proportion plus faible de BAD. De manière périphérique, on note aussi que ces deux programmes sur ce scénario particulier ont une proportion équivalente de KILLED. Une construction interne proche peut être la raison de cette égalité mais un nombre plus grand de scénarii doit être utilisé pour confirmer cette hypothèse.

Les attaques transientes

Après avoir présenté les effets des attaques permanentes, nous nous intéressons aux attaques transientes. On appelle attaque transiente une attaque qui n'est déclenchée qu'une fois pendant l'exécution, contrairement à une attaque qui est déclenchée à chaque fois que le bloc de code C contenant l'attaque est exécuté. Ainsi, le nombre d'attaques transientes par GOTO à tester (et à classifier) par ligne de code source est égal au nombre d'exécutions de l'instruction C correspondante. Nous avons testé exhaustivement ces attaques transientes et la figure 5.10 montre la classification des attaques pour la totalité du code de BZIP2. Le nombre de cas GOOD (resp. KILL) augmente (resp. diminue) étant donné que les attaques par saut arrière transientes ont une plus faible probabilité de provoquer une boucle infinie.

La figure 5.11 montre le classement des attaques pour la fonction mainQSort3 de BZIP2 en fonction du moment où l'attaque est déclenchée. Les attaques qui sont déclenchées la première fois que le bloc de code C correspondant est exécuté donne environ 30% de BAD, 30% de KILL, et 30% de GOOD. Les attaques déclenchées entre la troisième et la 25e exécution du bloc C

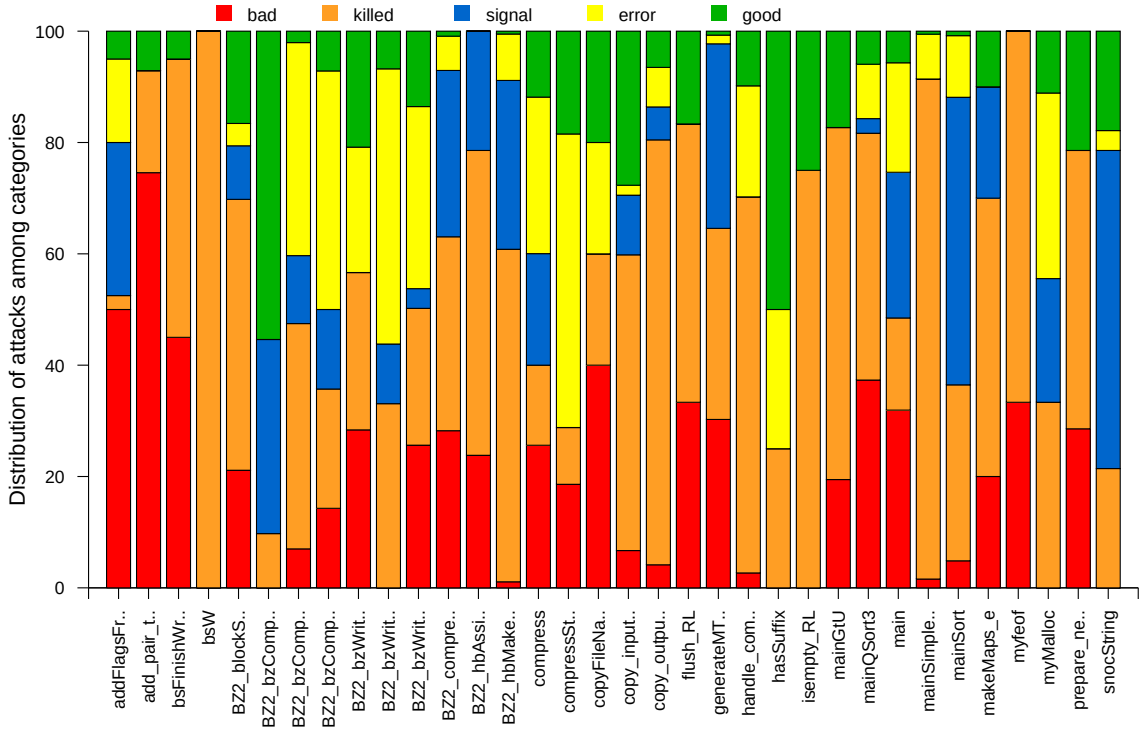


FIGURE 5.7 – "Génome" de BZIP2

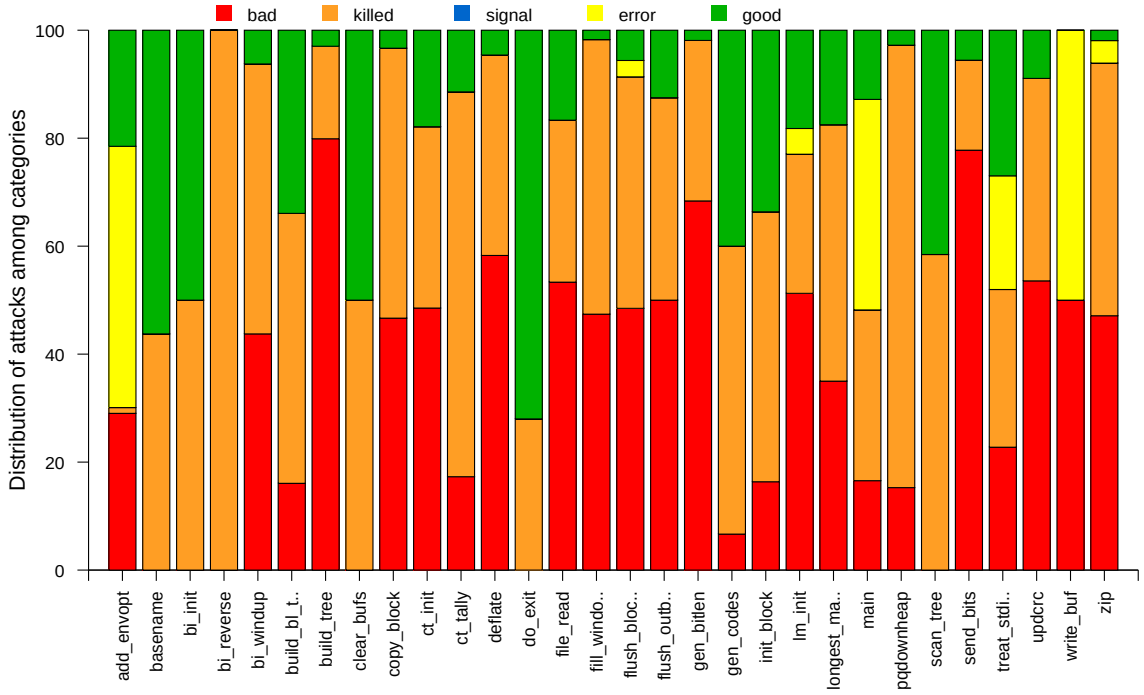


FIGURE 5.8 – "Génome" de GZIP

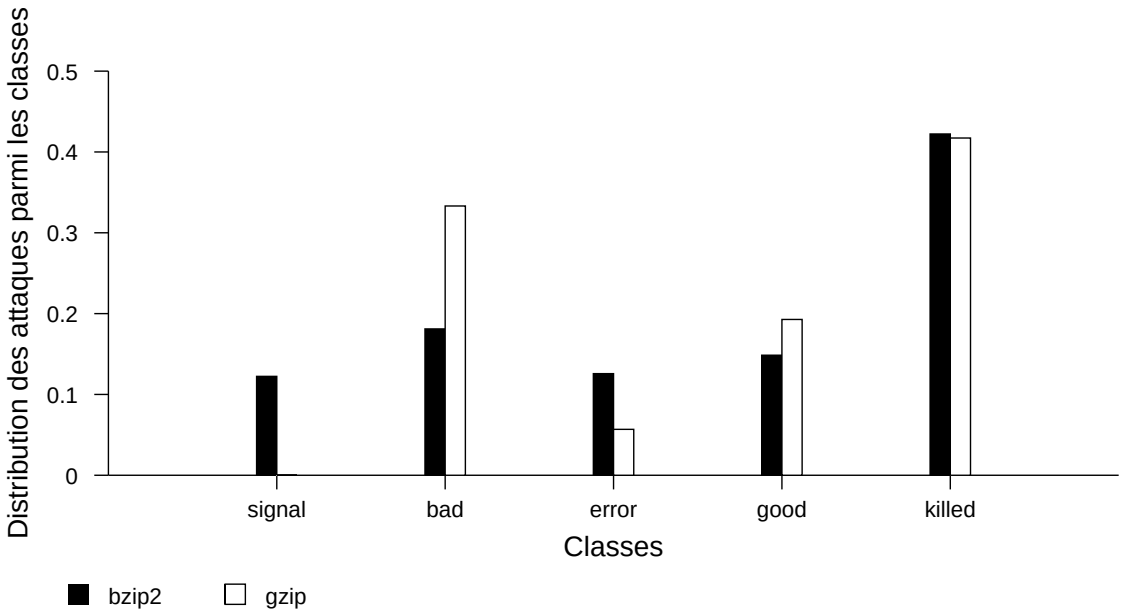


FIGURE 5.9 – Comparaison des "génomes" de BZIP2 et GZIP

réduisent le nombre de cas GOOD. Après la 26e exécution du bloc C, la figure 5.10 montre que le classement se stabilise. Cependant, cette stabilité dépend du nombre d'exécutions de chaque instruction C et des données d'entrée.

5.3.4 Couverture assembleur par le modèle C

Afin de valider le modèle d'attaque haut niveau établi dans le chapitre 3, nous détaillons dans cette section les aspects techniques des trois méthodes d'injection présentées précédemment. Le but de la section est de calculer expérimentalement le taux de couverture des attaques simulées au niveau assembleur par celles qui le sont au niveau C.

Le premier avantage qu'il y a à utiliser trois méthodes d'injection d'attaques différentes est que les méthodes assembleur et C peuvent être comparées afin de savoir si le modèle d'attaque C par GOTO donne une couverture satisfaisante des attaques possibles par saut en assembleur. De plus, l'utilisation d'un modèle de haut niveau nécessite de générer l'ensemble des binaires attaqués correspondant, ce qui entraine un grand nombre de générations de code source et de cycles de compilation. La méthode GDB injecte dynamiquement les attaques à haut niveau pendant l'exécution. Cette méthode réduit sensiblement le temps nécessaire à la réalisation des tests.

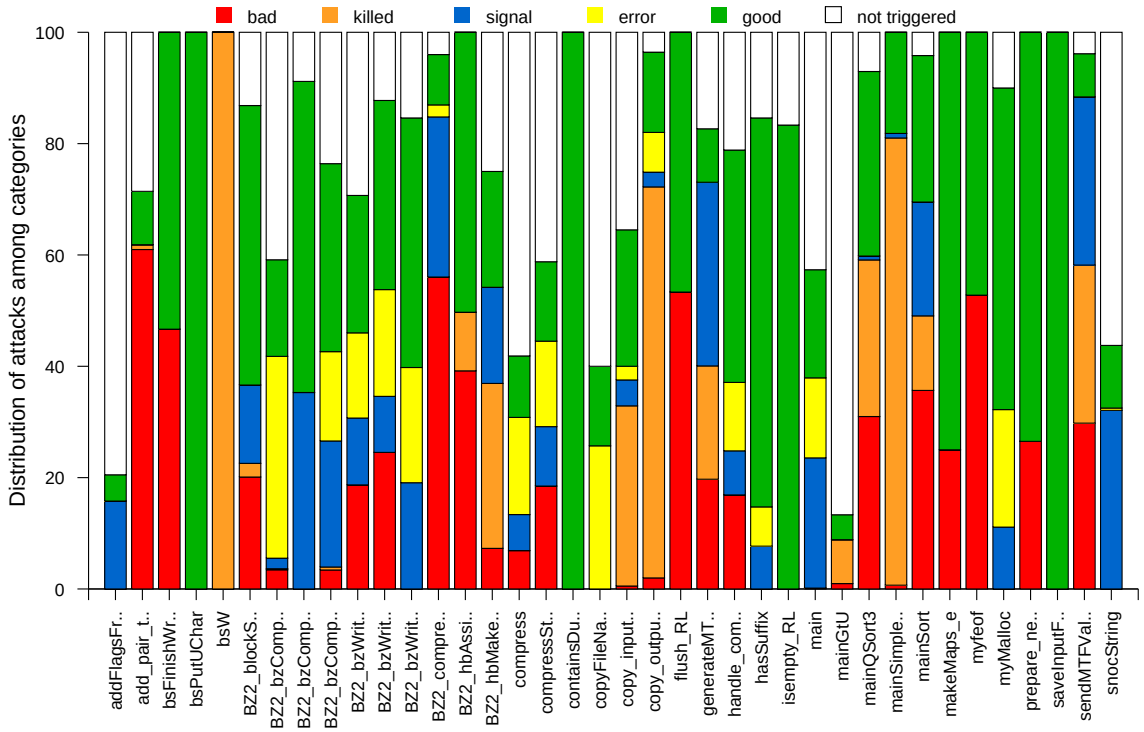


FIGURE 5.10 – Les attaques transientes injectées avec GDB dans la totalité de BZIP2

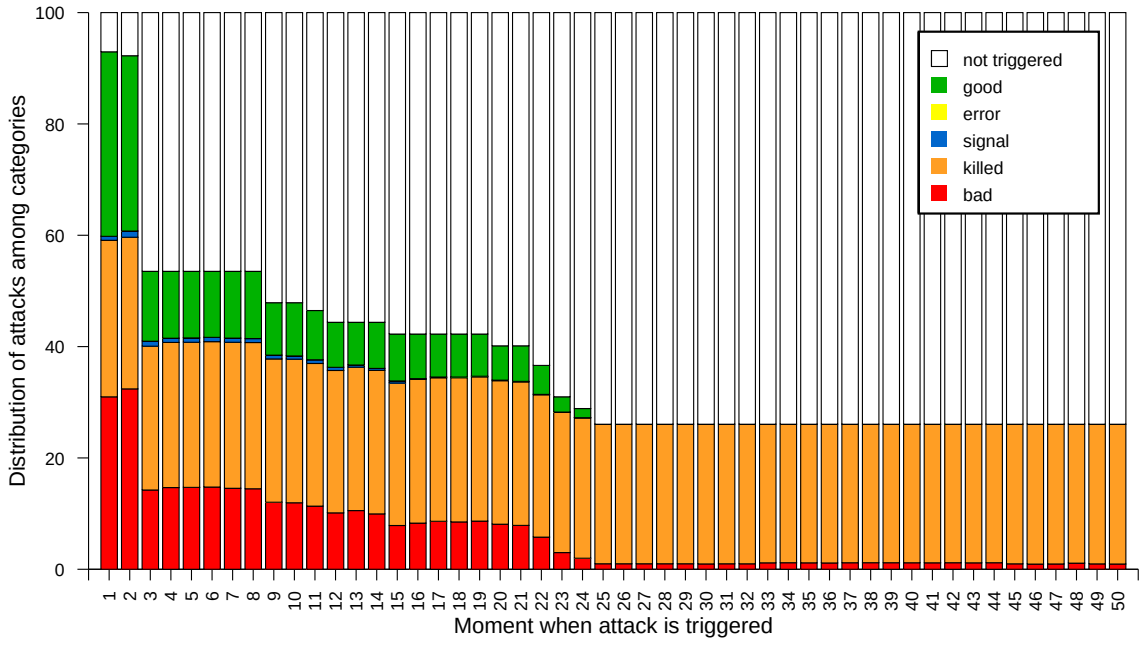


FIGURE 5.11 – Classement des attaques transientes pour la fonction mainQSort3 de BZIP2 en fonction du moment où l'attaque est déclenchée

Statistiques	ASM	C	gdb
Taille du code	26 103	8 643	8 643
Nb d'attaques	3 531 954	117 802	131 630
Temps Simu.	2d 18h	8h	2h
Nb BADs	273 129	14 050	5 417
Nb fichier vide	103 952	6 514	1 301
Uniq BADs	2 326	1 245	852
ASM couverture	100%	25%	25%

TABLE 5.1 – Statistiques des attaques simulées sur BZIP2

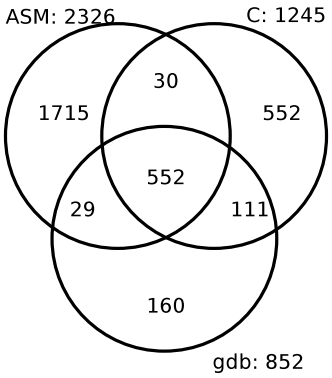


TABLE 5.2 – Couverture d'attaques

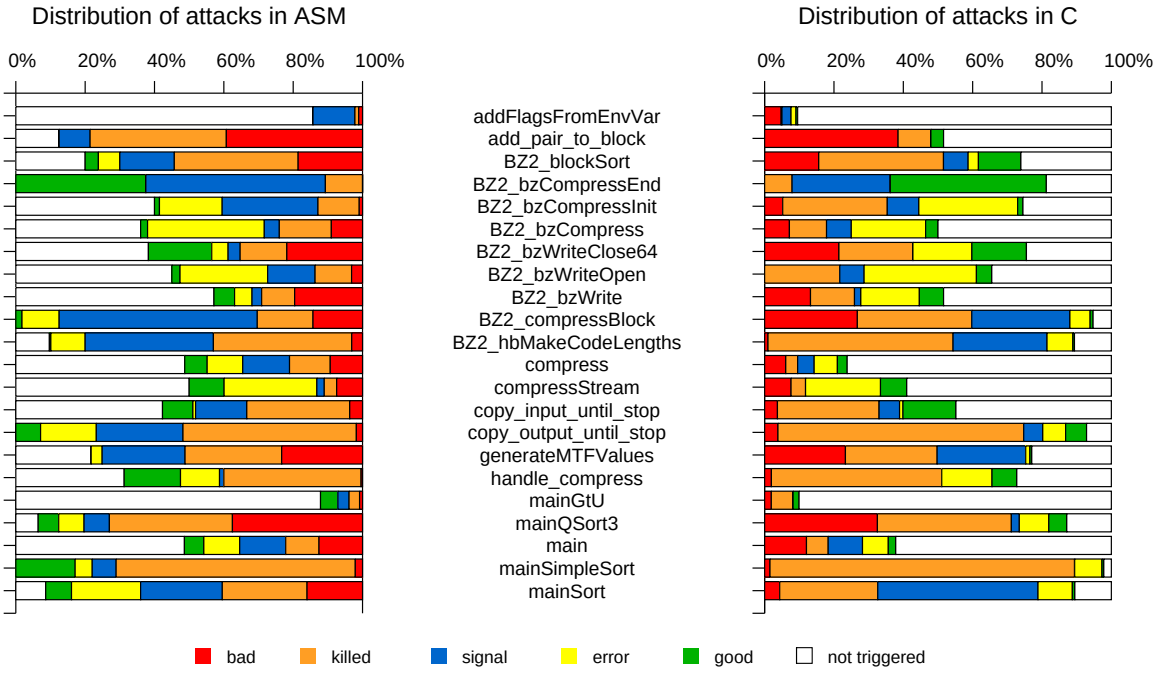


FIGURE 5.12 – Distribution des attaques permanentes injectée au niveau ASM / C pour les fonctions de BZIP2 de plus de 7 instructions

Nous proposons de comparer les conséquences des attaques du modèle d'attaque au niveau C aux conséquences des attaques au niveau assembleur. Ces dernières sont représentées dans la figure 5.2 en lignes pointillées. Nous comparons aussi nos résultats avec ceux obtenus grâce à une technique complémentaire basée sur l'injection d'attaques à l'aide d'un débogueur. Cette partie

de l'étude, dont les résultats sont données en section 5.3.4, est une vérification expérimentale de la méthodologie globale et n'est pas nécessaire pour utiliser la méthode proposée.

La figure 5.12 montre les classifications C et ASM (assembleur) des effets d'attaques permanentes sur des fonctions possédant plus de 7 instructions dans l'ensemble du code source de BZIP2. Les attaques par saut sont injectées pour chaque couple d'instructions assembleur. Cette figure montre que les distributions sont semblables ce qui suggère que le modèle d'attaque à haut niveau permet de couvrir une portion significative du modèle à bas niveau. Les fonctions sensibles, qui ont une grande proportion de BADs, doivent être analysées de manière approfondie, comme présenté plus loin dans la section 5.4.2.

Afin de comparer les taux de détection des attaques réussies de la méthodologie d'injection d'attaque, nous avons calculé l'identifiant MD5 de chaque fichier compressé par une exécution du programme. Si le MD5 correspond à celui calculé en utilisant une version originale du programme BZIP2, le scénario de test est considéré GOOD, sinon il est BAD.

Les tableaux 5.1 et 5.2 donnent les résultats statistiques obtenus avec les trois méthodologies. D'abord, on remarque qu'il y a 30 fois moins de cas de test pour la méthodologie C par rapport à la méthodologie ASM ce qui se traduit par une exécution de l'ensemble des tests nécessitant 8 fois moins de temps. La méthodologie GDB a un comportement similaire et est encore plus rapide car elle ne nécessite pas de nouvelle compilation de l'ensemble du programme et factorise une partie de l'exécution des tests. Ensuite, nous calculons le nombre de fichiers compressés différents obtenus avec le programme attaqué. Chaque fichier correspond à un scénario unique BAD et nous considérons que deux attaques qui retournent le même fichier ont le même effet. Un cas particulier du scénario BAD apparaît quand un fichier vide est obtenu ce qui arrive dans près de 46% des scénarii pour la méthodologie C. Comparer les cas uniques de BAD permet de déterminer la couverture d'une méthodologie par une autre. L'intersection précise des groupes obtenus est donné dans la tableau 5.2. Pour la méthodologie C, nous obtenons 1245 BADs uniques dont 482+21 sont partagés avec la méthodologie ASM qui totalise 2326 BADs. Ces résultats nous amènent à une couverture de 21% des résultats obtenus avec la méthodologie assembleur par les résultats obtenus avec la méthodologie C. La méthodologie GDB confirme ces résultats.

5.4 Analyse des résultats

5.4.1 Distribution des attaques par taille du saut

La figure 5.13 montre la distribution d'attaques par taille du saut (nombre de lignes de code successives sautées par ce saut). La courbe GOOD montre que le programme est particulièrement sensible aux attaques par saut de grande taille (supérieur à 260 lignes) : le programme BZIP2 ainsi attaqué ne retourne pas de fichier compressé correct. On remarque de manière plus étonnante que certains saut arrière mènent à des GOOD. Ces attaques ne sont pas classées comme KILL car le programme n'est pas entré dans une boucle infinie. La courbe BAD montre que la plupart des attaques réussies ont une taille de saut entre 10 et 20 lignes de code. Le nombre de ces attaques réussies décroit ensuite d'une manière quasiment linéaire. D'une manière plus surprenante, le nombre d'attaques classifiées comme BAD est quasiment constant pour les attaques par saut arrière jusqu'au sauts arrière de -540 lignes où il tombe à 0.

D'un point de vue sécuritaire, cette figure aide à identifier les plages d'attaques qui peuvent être dangereuses pour le programme considéré. Pour BZIP2 les scénarii BAD sont des sauts en

avant entre 0 et 450 lignes de code. Ce résultat peut être différent pour un autre code source.

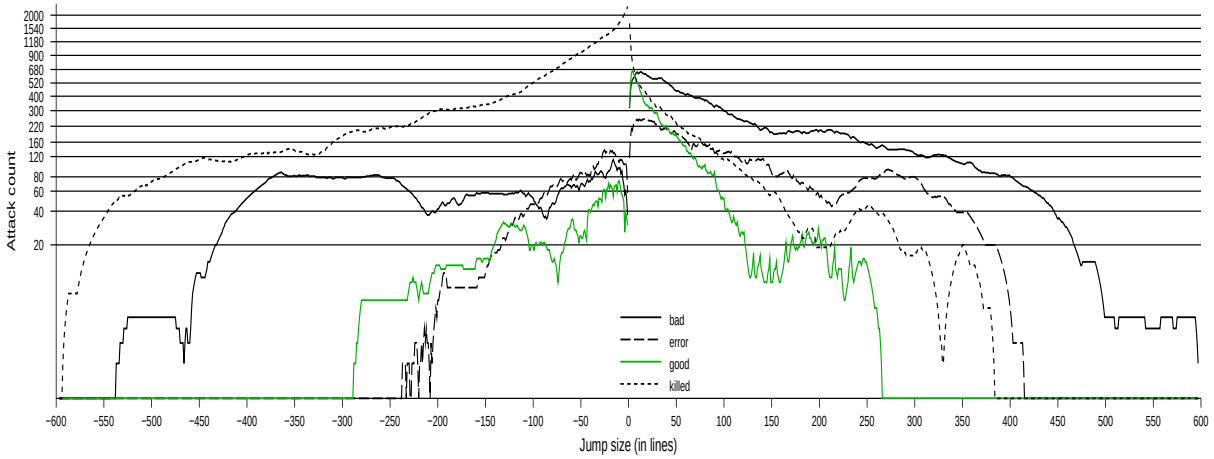


FIGURE 5.13 – Nombre d'attaques en fonction de la taille du saut exprimé en lignes de code C

5.4.2 Analyse visuelle des fonctions vulnérables

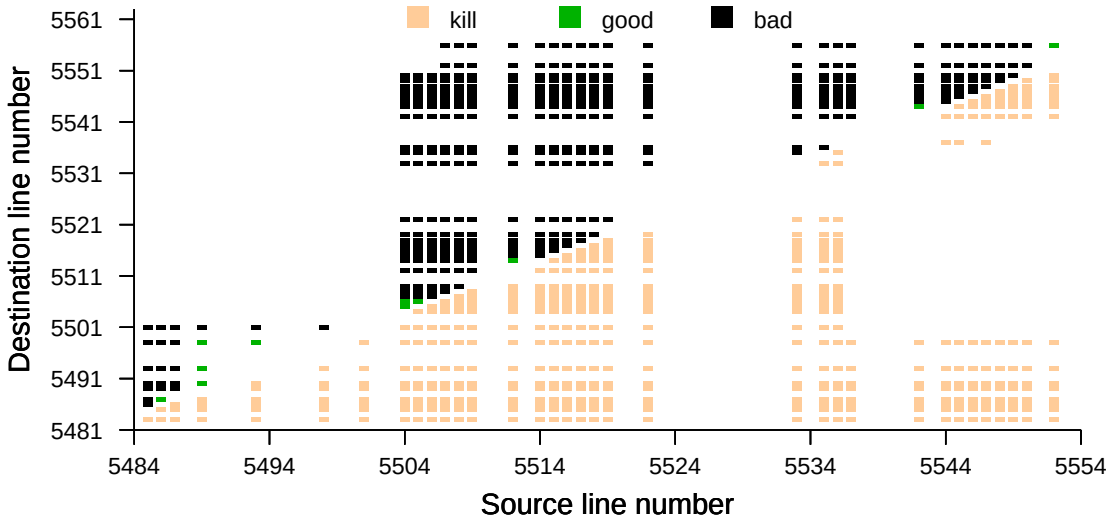


FIGURE 5.14 – Classification spatiale des attaques good/bad/kill en fonction des lignes source et destination, simulées en C pour la fonction BZ2_compressBlock

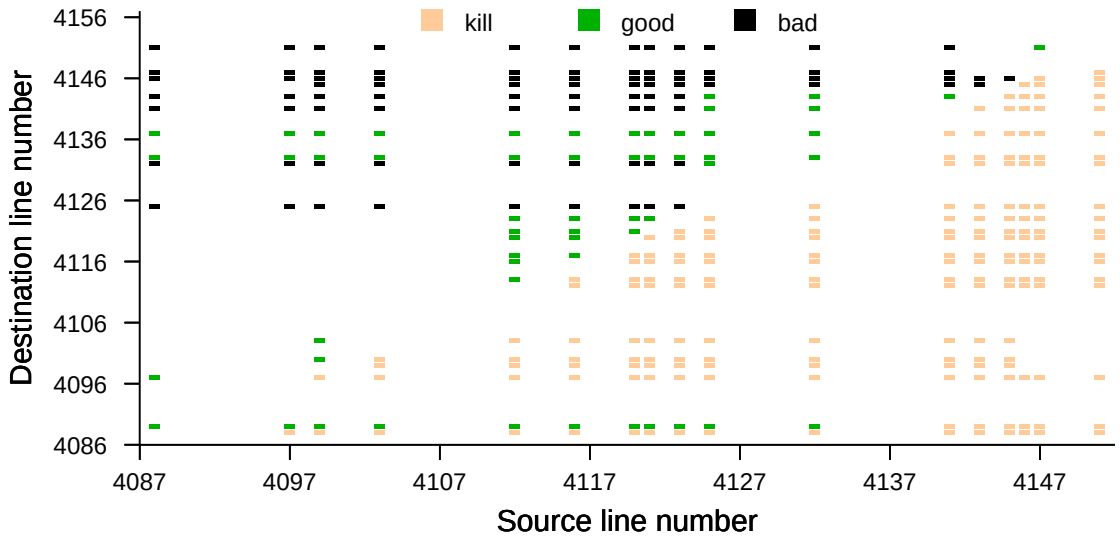


FIGURE 5.15 – Classification spatiale des attaques good/bad/kill en fonction des lignes source et destination, simulées en C pour la fonction BZ2_blockSort

Les résultats sont obtenus par l'exécution d'un scénario fonctionnel. Par construction, la classification des résultats prend en compte le gain que peut obtenir l'attaquant en provoquant une modification de l'exécution avec une faute. Les fonctions ayant des résultats classées comme BAD sont donc sensibles d'un point de vue de la sécurité, par rapport au scénario exécuté. Elles représentent la possibilité qu'un attaquant obtienne un gain sans que le système ne s'en rende compte et ne déclenche de contre-mesure.

Une fois qu'une fonction sensible et vulnérable aux fautes est identifiée, grâce à son taux important de cas BAD, le développeur chargé de la sécurité a besoin d'une représentation plus précise des attaques dangereuses afin de comprendre pourquoi ces attaques réussissent. Les figures 5.14 et 5.15 montrent deux exemples d'une représentation spatiale des classes d'attaques GOOD, KILL et BAD (les autres classes ne sont pas représentées). Nous utilisons les conventions de la figure 5.3. Comme on pouvait s'y attendre, les sauts arrières génèrent un grand nombre de cas KILL tandis que les sauts en avant amène à des cas BAD.

Pour la fonction BZ2_compressBlock de la figure 5.14, un grand nombre de cas BAD sont obtenus si la source de l'attaque se trouve entre les lignes de code source 5504 et 5550. Pour la fonction BZ2_blockSort, le nombre de cas BAD est moins important. On remarque aussi qu'une série d'attaques réussit si la ligne de code source de destination de l'attaque se trouve après la ligne 4126. Ces informations sont très utiles pour le développeur chargé d'implémenter des contre-mesures adaptées. Nous développons ce dernier point à l'aide de l'exemple de la fonction BZ2_blockSort dans la prochaine section.

Listing 5.2– Implémentation d'une contre-mesure pour la fonction BZ2_blockSort ne nécessitant pas de décalage de la numérotation des lignes du code de départ

```
/** Code original **/                                    /** contre-mesure **/
                                  /** kc(); = {perror("KILLCARD");exit(-1);} **/
                                                          int security = 48;
...
                                                                  security++;
s->origPtr = -1;
                                          if (security != 49) kc(); security++;
for (i = 0; i < s->nblock; i++)
{                                     if (security != 50+2*i) kc(); security++;
    if (ptr[i] == 0) {
        s->origPtr = i;
        break;
    }                                 if (security != 51+2*i) kc(); security++;
}
                                                     if (security < 50) kc();
AssertH(s->origPtr != -1, 1003);
```

5.4.3 Implémentation de contre-mesures

Le listing 5.2 montre un extrait de la fonction BZ2_blockSort dans lequel la partie gauche du listing est le code original. Étant donné que nous avons découvert un grand nombre de BAD dans la figure 5.15, nous proposons une implémentation d'une contre-mesure naïve pour sécuriser la zone 4140-4151 . Cette contre-mesure, sur la partie droite du listing 5.2, consiste à incrémenter et tester régulièrement un simple compteur.

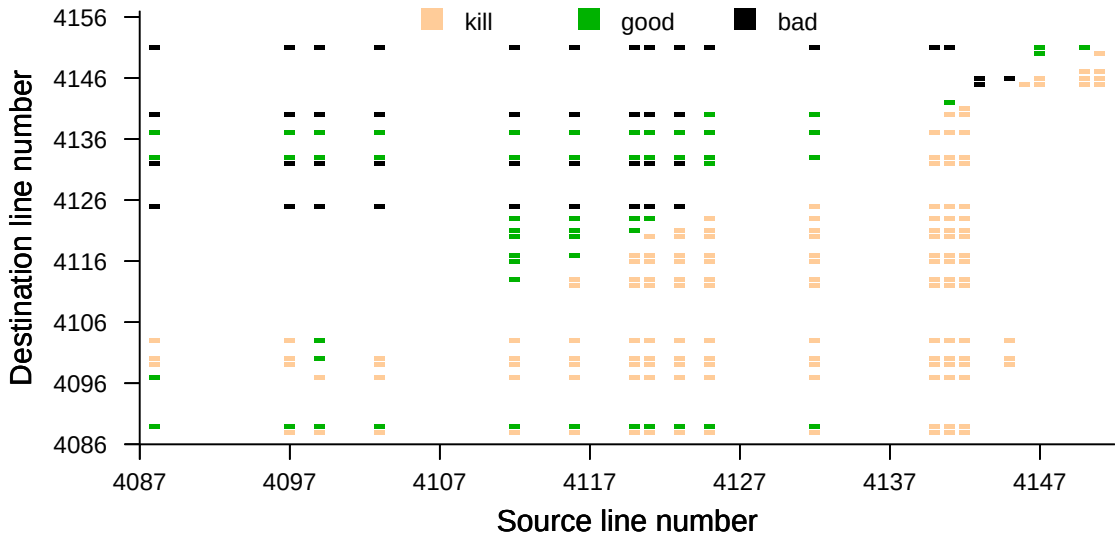


FIGURE 5.16 – Classification spatiale des attaques good/bad/kill en fonction des lignes source et destination, simulées en C pour la version sécurisée de BZ2_blockSort

La figure 5.16 montre la nouvelle classification des attaques. Plusieurs cas BAD ont disparu pour être remplacé par des cas ERROR (les carrés noirs ont disparu de la partie supérieure de

la figure). Certains cas KILL ont aussi disparu de la même zone (partie droite de la figure). Cet exemple montre comment un développeur peut implémenter une contre-mesure permettant de contrer un grand nombre d'attaques par contrôle de flot réalisées dans la pratique par un saut dans le code depuis le début de la fonction jusque dans la zone 4140-4151.

5.5 Résultats expérimentaux sur code de carte à puce

Afin d'implémenter un oracle générique permettant de classer les résultats des exécutions de codes de cartes à puce, nous avons implémenté un oracle externe à la carte, i.e., dans un terminal de simulation. A l'aide de ce système, un statut d'erreur signalé par la carte au terminal se traduit par un cas BAD. L'avantage principal résultant de la création d'un tel oracle est qu'il peut être utilisé pour n'importe quelle fonction impliquée dans une opération qui retourne un statut au terminal.

Comme mentionné précédemment, la signification d'un cas BAD peut changer suivant l'application. Ce cas représente un comportement sécuritaire incorrect pour le système. Ce comportement sécuritaire incorrect se manifeste par des évènements observables à un certain niveau du système. Dans notre cas, nous avons choisi, pour des soucis de généricité, de nous intéresser au statut renvoyé par la carte car il correspond à une philosophie de test en boîte noire proche de ce que peut réaliser un attaquant. Cependant, il est aussi possible, pour plus de précision, de se placer dans une situation de test en boîte blanche au niveau du code source du programme. Dans ce cas, on peut, par exemple, créer un oracle capable d'apprécier si une écriture mémoire est correctement effectuée, qu'une variable est bien incrémentée (respectivement décrémentée) ou qu'une valeur discrète est bien affectée à une variable. Cette notion d'oracle peut facilement être étendue puisqu'on peut aussi créer à un autre niveau un oracle capable d'apprécier si lors d'une exécution le composant laisse fuir de l'information par canaux cachés. En fait, pour toute garantie de sécurité souhaitée (une propriété de sécurité dans notre cas), s'il est possible de déterminer un ou plusieurs critères observables reflétant le respect ou non de la propriété, on peut créer un oracle qui, à partir de ces critères, différencie un comportement correct d'un comportement qui ne l'est pas. Ainsi, on peut différencier les attaques réussies des autres. En variant et en combinant les oracles il est possible de prendre en compte l'ensemble des capacités d'observation et d'action des attaquants.

La méthodologie d'injection au niveau du code source a été appliquée à une fonction sensible d'un projet d'Oberthur Technologies réalisant une authentification basique. Le scénario considéré consiste à ce qu'un utilisateur fournisse de mauvais certificats au mécanisme d'authentification, ce qui doit conduire la carte à refuser d'authentifier celui-ci. L'attaquant cherche à obtenir cette authentification en attaquant physiquement la carte.

La fonction considérée, composée de 180 lignes de code C, est responsable de la partie principale du processus d'authentification. D'un point de vue sécuritaire, toute attaque détectée sur une carte à puce doit déclencher une réponse sécuritaire consistant à "tuer" la carte. La catégorie KILLCARD est la réponse sécuritaire la plus forte que peut déclencher la carte quand une attaque est détectée. Les cas où une erreur ou un signal est retourné par la carte ou lorsque la décision finale est de refuser l'accès à l'utilisateur (GOOD) ne sont pas des cas dangereux pour la sécurité de la fonction car l'attaque a échoué. Par conséquent, les attaques qui permettent l'authentification sous attaque sont celles qui retournent un jeton valide et un statut valide au terminal. Nous

avons configuré l'oracle pour qu'il classe ces cas dans la catégorie BAD incluant donc les attaques considérées comme dangereuses pour la sécurité du programme. Notons que pour améliorer les performances de nos campagnes de test, nous catégorisons d'abord les résultats suivant le statut retourné par la carte au terminal et analysons la validité du jeton d'authentification que dans un second temps.

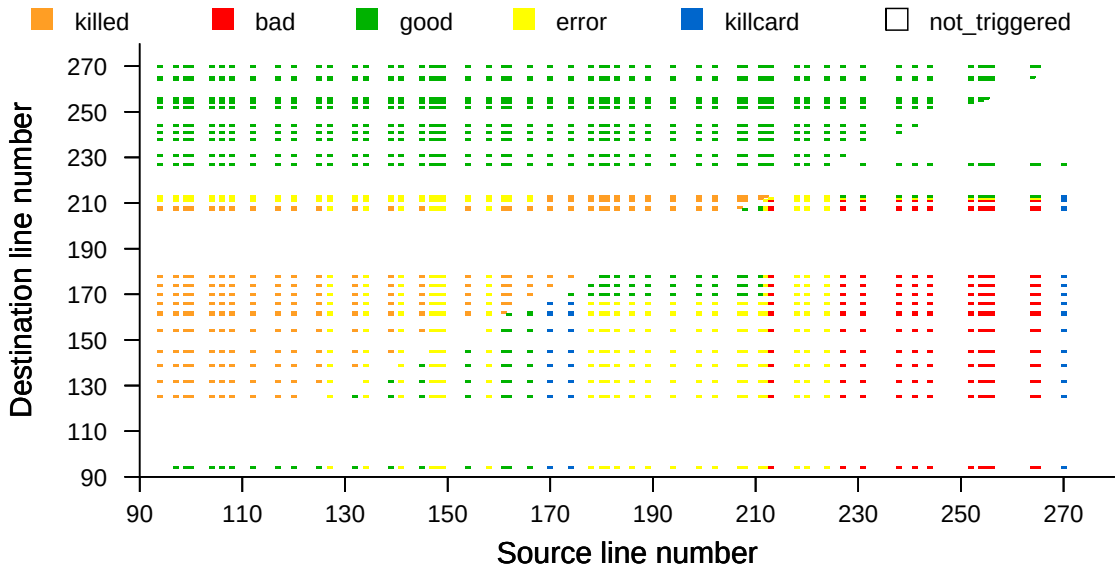


FIGURE 5.17 – Classification spatiale des attaques kill/bad/error en fonction des lignes source et destination, simulées en C pour une fonction sensible du code source de d'une carte à puce

La figure 5.17 présente une classification spatiale des résultats des attaques. Dans ce scénario particulier, les cas BAD pour cette fonction apparaissent quand le saut a lieu dans une zone déterminée par les lignes (212-268,92-209). Les points obtenus dans cette zone représentent des attaques potentiellement dangereuses car elles ont le comportement d'une attaque réussie et nécessitent uniquement un saut dans le code pour provoquer ce comportement.

La distribution de ces attaques considérées comme dangereuses est représenté dans l'espace dans la figure 5.18. Cette figure montre la relation entre le nombre d'attaques considérées comme dangereuses et le nombre de lignes du code source original qui sont sautées. Pour cette fonction sensible stimulée par ce scénario de test particulier, il apparaît que les sauts d'une distance de 40 à 120 lignes de code successives sont celles qui donnent le plus d'attaques réussies.

Étant donné que, pour Oberthur Technologies, seuls les sauts d'une distance faible sont considérés comme des attaques réalistes car elles sont réalisables dans la pratique de manière contrôlée et reproductible, nous nous sommes intéressés à 3 attaques d'une distance de saut inférieure à 10 lignes de code C successives. Nous avons examiné les 3 attaques identifiées et avons déterminé 2 faux positifs, i.e., où le statut renvoyé par la carte est correct mais où le jeton d'authentification généré est incorrect. Par conséquent, seul un cas sur ces 3 cas BAD est vraiment dangereux. En utilisant les lignes source et destination identifiant le saut concerné, nous sommes

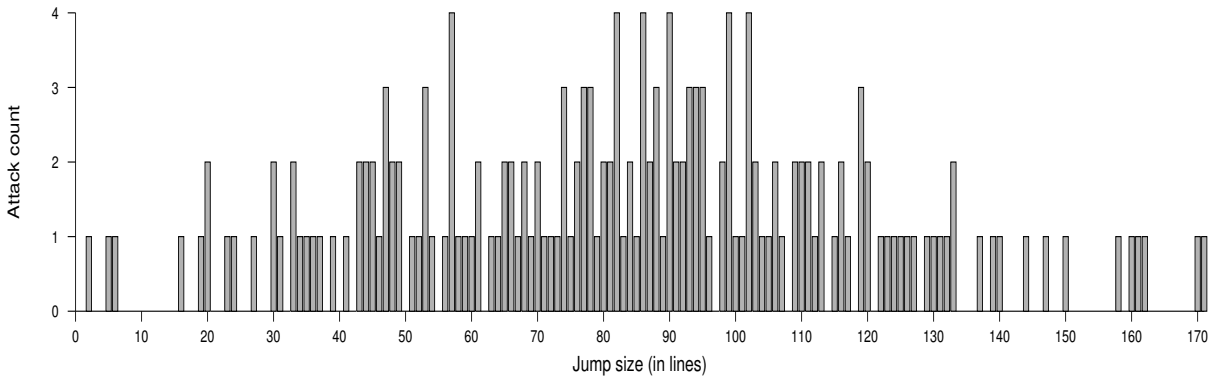


FIGURE 5.18 – Nombre de résultats BAD en fonction de la distance de l'attaque par saut

remonté à la cause de la vulnérabilité : un saut direct dans une branche fonctionnelle incorrecte d'une condition. La conséquence fonctionnelle de cette attaque est une authentification réussie par l'utilisateur avec des certificats incorrects.

Cette attaque a été identifiée comme la plus réaliste vis-à-vis du processus d'authentification. Les experts sécuritaires ont néanmoins rapporté que la distance du saut assembleur considéré rendait l'attaque extrêmement difficile à réaliser et qu'une contre-mesure supplémentaire n'était pas nécessaire.

La figure 5.18 montre que si des sauts d'une distance plus grande pouvaient être contrôlés précisément, le taux de réussite des attaques serait considérablement amélioré et l'implémentation considérée deviendrait vulnérable à des attaques exploitant une perturbation du contrôle de flot au niveau fonctionnel. Pour terminer, on peut préciser que le critère discriminant de la catégorie BAD peut être amélioré afin d'incorporer la vérification du jeton d'authentification. Cette amélioration permettrait de différencier les attaques vraiment dangereuses et ainsi d'exclure les faux positifs identifiés. Cependant, pour ne pas pénaliser les performances des campagnes de test, cette vérification doit être faite seulement après avoir vérifié que le statut retourné est identique au statut d'un scénario correct.

5.6 Conclusion

Dans ce chapitre, nous avons proposé une méthodologie de test dynamique ainsi qu'un environnement de test afin de valider le comportement fonctionnel d'une implémentation sous attaque physique. Cette méthodologie est une alternative à la solution statique de vérification d'intervalles de valeur proposée dans le chapitre précédent. Elle permet d'obtenir une synthèse du comportement sous attaque sur l'ensemble du code stimulé par un cas de test. Les attaques simulées correspondent au modèle d'attaque établi dans le chapitre 3. L'ensemble des attaques par saut du modèle d'attaque par GOTO peuvent ainsi être simulées. À partir d'une représentation visuelle du comportement obtenu, le développeur peut isoler les fonctions susceptibles d'engendrer une faille fonctionnelle de sécurité. Cette représentation permet également de filtrer les attaques pouvant être considérées comme réalistes jusqu'à obtenir un échantillon réduit de lignes de code à considérer. Le développeur doit confirmer la faisabilité d'une attaque maté-

rielle à ces endroits du code source en prenant en compte l'architecture matérielle utilisée. Cette étape consiste à vérifier si le code assembleur généré est également sensible à des attaques du modèle d'attaque. Pour ce faire, une des méthodologies assembleur proposées peut être utilisée. Seule la méthodologie au niveau C a été implémentée dans un contexte industriel. Celle-ci présente l'avantage de pouvoir être intégrée au processus de développement et utilisé alors que le composant cible et donc la plateforme matérielle associée n'est pas encore disponible. Un autre avantage est que cette méthode contrairement à la méthode du chapitre précédent ne nécessite pas d'annotation pour fournir de résultats. L'ensemble du processus est automatisé. La génération de l'ensemble des codes attaqués permet également de rejouer une attaque présente dans une campagne d'attaque afin de recréer le contexte d'exécution de l'attaque. Cette méthode couvre l'ensemble des attaques fonctionnelles par modification du contrôle de flot du programme.

L'expérience pratique montre la faisabilité de la méthodologie et valide celle-ci à l'échelle d'un programme. Nous avons ainsi réussi à implémenter notre technique d'injection d'attaque sur BZIP2 et GZIP. En analysant les résultats expérimentaux obtenus, on peut dire que le modèle d'attaque C simulant les conséquences bas niveau d'une attaque physique couvre une partie significative de ces attaques. La méthodologie proposée permet d'identifier l'ensemble des attaques potentiellement dangereuses au niveau fonctionnel correspondant au modèle d'attaque établi. La méthodologie permet au développeur de facilement remonter jusqu'à l'origine fonctionnelle de l'attaque et aussi l'aide à implémenter et tester une contre-mesure adéquate. En utilisant l'outil développé, un développeur peut aussi tester de manière itérative l'implémentation de ses contre-mesures jusqu'à ce qu'elles deviennent efficaces tout en gardant la garantie qu'aucune attaque ne compromet son implémentation.

5.7 Perspectives

Cette méthodologie peut servir dans des travaux futurs afin d'injecter automatiquement des contre-mesures au niveau du code source du programme. En effet, la phase d'instrumentation d'attaque peut être dérivée afin d'inclure des contrôles de sécurité ou des contre-mesures. Les auteurs de [Akkar *et al.* 2003] présentent une solution similaire.

La méthodologie proposée permet à partir d'un test existant de valider son comportement en cas d'attaque. Il pourrait être intéressant de générer automatiquement ces tests en déterminant les vecteurs d'entrée possibles permettant de simuler la portion de code souhaitée. Les auteurs de [Nori *et al.* 2009] adoptent une telle approche en associant l'analyse statique, pour la découverte de nouveaux vecteurs de test, associée à des tests dynamiques et un algorithme de raffinement successif.

La simulation d'attaques multiples est possible grâce à la méthodologie proposée. Il serait intéressant de caractériser à nouveau le code en stimulant des doubles fautes afin de caractériser les formes de code sensibles à ce type d'attaques. En effet, ces attaques ont vocation à se généraliser avec la montée en compétence des attaquants. Disposer d'une caractérisation du code vis-à-vis de ces attaques peut aider dans le processus de sécurisation du code et la création de nouvelles contre-mesures.

Chapitre 6

Outils industriels

Sommaire

6.1 Introduction . . . 169
6.2 Environnent et facteurs . . . 170
6.2.1 Environnement industriel . . . 170
6.2.2 Processus de vérification de la sécurité . . . 170
6.2.3 Facteurs à prendre en compte . . . 170
6.3 Outils et méthodes existantes . . . 171
6.3.1 Domaines similaires . . . 171
6.3.2 Cas particulier de la sécurité . . . 173
6.3.3 Techniques statiques et dynamiques . . . 173
6.3.4 Sécurité et absence d'erreurs . . . 173
6.4 Réalisations techniques . . . 175
6.4.1 Amélioration de la relecture de code . . . 175
6.4.2 Amélioration de la tracabilité fonctionnelle . . . 175
6.4.3 Vérification de formes de code . . . 176
6.4.4 Injection simple et exhaustive d'attaques . . . 177
6.4.5 Analyse et interrogation interactive du code source . . . 181
6.5 Conclusion . . . 186
6.6 Perspectives . . . 187

6.1 Introduction

Dans ce chapitre, nous présentons les réalisations complémentaires effectuées au cours de cette thèse. Ces réalisations répondent de manière outillée à des besoins en termes de sécurisation de code source dans un environnement industriel. Tout d'abord, en section 6.2, nous donnons quelques éléments de contexte afin de mieux cerner les raisons qui ont mené à la réalisation de tels outils. Puis, en section 6.3, nous présenterons différentes méthodes et outils existants permettant de résoudre des problèmes de sécurité. Finalement, dans la section 6.4, nous présenterons les solutions techniques réalisées dans le cadre industriel au cours de cette thèse qui contribuent à améliorer le processus de sécurisation de code embarqué. Les contributions couvertes dans ce chapitre font référence aux publications [Chambérot & Kauffmann-Tourkestansky 2009] et [Berthomé *et al.* 2011]. Ces deux contributions sont d'ordre industriel en montrant comment des méthodologies élaborées dans les contributions précédentes ont pu être mises en œuvre dans un environnement industriel.

6.2 Environnent et facteurs

Avant de présenter les réalisations techniques développées dans le cadre industriel au cours de cette thèse, il convient de donner un aperçu de l'environnement industriel embarqué dans lequel cette thèse s'est déroulée.

6.2.1 Environnement industriel

Tout d'abord, présentons quelques métriques sur les projets embarqués évoqués dans cette thèse. Ces projets représentent l'ensemble du code source du système d'exploitation d'une carte à puce et de ses applications. Ils sont composé d'environ 100 000 à 200 000 LOC programmés en langage C. Les projets Java Card ne sont pas abordés dans cette thèse et aucune réalisation technique n'a été effectuée dans ce cadre. Cependant certaines techniques génériques mises au point peuvent être adaptées à d'autres langages et des environnements de développement différents.

L'environnement de développement dans le cadre de cette thèse est Microsoft Visual Studio 2005. Dans une première phase de développement, un projet sous forme de DLL est d'abord créé et interfacé avec une solution de simulation de terminal propriétaire. Le projet sous cette forme permet des tests et des corrections sur plusieurs itérations du cycle de développement. Une fois le projet avancé, dans une deuxième phase de développement, ce projet est porté sur l'environnement propriétaire du fondeur (en utilisant KEIL ou un autre environnement de développement) et de nouvelles itérations du cycle de développement sont faites en parallèle avec ces deux projets. Finalement, un binaire final est livré au fondeur qui l'embarque sur le composant réel.

6.2.2 Processus de vérification de la sécurité

Dans ce processus de développement, la vérification de la sécurité se présente sous la forme de plusieurs relectures de code faisant intervenir le développeur qui a implémenté la solution logiciel et l'expert en sécurité. Le premier a la connaissance des besoins fonctionnels de l'application sur laquelle il travaille et donc du code qu'il a développé ; le second est à jour vis-à-vis des dernières attaques physiques existantes et des chemins d'attaques associés. Des tests sécuritaires sont aussi réalisés dans la deuxième phase de développement pour s'assurer que les contre-mesures mises en place sont bien déclenchées lors de la détection d'une attaque physique. Ces tests demandent qu'un développeur débogue son application et modifie manuellement au moment de l'exécution une instruction pour simuler une attaque.

Avec les réalisations techniques de cette thèse, un des buts est d'améliorer le processus de sécurisation mis en place tout en prenant en compte les facteurs industriels existants.

6.2.3 Facteurs à prendre en compte

Plusieurs contraintes industrielles ont été prises en compte lors de la création ou l'intégration d'outils visant à améliorer le processus de sécurisation.

Les facteurs environnementaux

Des facteurs environnementaux ont conditionné des choix sur les outils à mettre en œuvre dans le cadre industriel de cette thèse. Tout d'abord l'environnement propriétaire du fondeur ne peut pas être modifié. Celui-ci est basé sur un compilateur propriétaire et spécifique au composant utilisé. Par conséquent, toute sécurisation basée sur une modification du compilateur n'est pas envisageable ; tout comme la modification du binaire a posteriori de la compilation. En cas de découverte d'une faille sécuritaire, il aurait été difficile de distinguer si l'origine du problème se situait dans le code d'origine ou le code rajouté par le compilateur. Une telle solution est également difficile à maintenir. Ces contraintes ont poussé à la création d'outils intervenant avant la compilation c'est-à-dire sur le code source C du programme.

Des contraintes liées au contexte embarqué sont aussi entrées en jeu dans la conception des outils d'aide à la sécurisation. Parmi celles-ci, le haut niveau d'optimisation d'un code embarqué (par un compilateur propriétaire) rend difficile la rétro ingénierie ou des formes de décompilation de code assembleur. De plus, des contraintes liées à la réutilisation de code existant ainsi que la rapidité de développement demandé dans un marché concurrentiel ont affecté le cahier des charges des outils développés.

Les facteurs techniques

En plus de ces facteurs environnementaux, différents facteurs techniques sont aussi intervenus. Ces facteurs techniques se résument essentiellement à des considérations de passage à l'échelle d'outils existants ou de méthodes de sécurisation. Un autre facteur important est l'équilibre entre la pertinence des résultats fournis et l'utilisabilité des outils.

Si un grand nombre de techniques de résolution existent dans la littérature, toutes n'ont pas encore été adaptées de manière viable dans un contexte industriel.

Les facteurs humains

L'utilisabilité d'un outil repose sur les utilisateurs de celui-ci. Certains paramètres entrent en jeu comme la courbe d'apprentissage, la charge de travail nécessaire afin d'obtenir des résultats pertinents. Ces paramètres se traduisent en une réticence vis-à-vis de l'adoption de l'outil dans les habitudes de travail dans un environnement industriel.

L'ensemble de ces facteurs a été pris en compte dans les réalisations techniques effectuées en milieu industriel au cours de cette thèse. Dans la majeure partie des cas, un équilibre a été trouvé afin de mitiger l'effort et d'amener progressivement vers des résultats pertinents.

6.3 Outils et méthodes existantes

6.3.1 Domaines similaires

Différentes techniques permettant une validation ou une vérification de la sécurité existent.

La validation implique l'utilisation d'un oracle qui juge en fonction de l'observation d'un résultat obtenu si une garantie souhaitée est obtenue. Il s'agit bien souvent de résultats pratiques obtenus lors de tests.

La vérification quant à elle implique une modélisation faisant apparaitre la totalité des éléments (et uniquement ceux-ci) entrant en jeu dans la résolution d'une garantie ainsi que la résolution effective de cette garantie dans tous les scénarii possibles envisagés. On obtient ainsi une preuve de la garantie souhaitée. Il s'agit dans ce cas bien souvent d'une preuve au sens mathématique.

Ce travail réalisé dans cette thèse touche plusieurs secteurs concernés par l'effet que peut provoquer une faute sur le fonctionnement de leur système. Le contexte d'utilisation, les priorités fonctionnelles et les conséquences sécuritaires ne sont cependant pas les mêmes. Les secteurs électrique et électronique cherchent à assurer le bon fonctionnement de leur système afin d'assurer une continuité de service. En aéronautique, les systèmes ne sont pas confrontés à des attaquants malicieux mais à des facteurs environnementaux extrêmes ; par contre la panne de fonctionnement d'une partie ou la totalité d'un système peut avoir des conséquences désastreuses comme la perte de vies humaines ce qui implique une recherche extrêmement poussée des erreurs de fonctionnement voire une preuve de l'absence de ceux-ci. Finalement en embarqué, l'attaquant est malicieux et cherche à modifier le comportement du code afin de l'avantager dans une situation où il ne doit pas l'être.

On retrouve des besoins similaires dans plusieurs autres domaines de recherche :

- les codes correcteurs d'erreur ;
- le test de programmes ;
- la sécurité logicielle et contre-mesures associées ;
- le Model Checking ;
- l'analyse et vérification de programme.

Les codes correcteurs sont utilisés afin de garantir une continuité de fonctionnement dans des conditions où le système peut être amené à transmettre un signal dont l'intégrité peut être compromise. Dans le cas de la sécurité embarquée le flot d'exécution est visé par un attaquant. Le test des programmes joue ici aussi un rôle majeur dans la validation des programmes embarqués. En effet, tester en situation réelle est souvent le seul moyen d'avoir une confirmation *sur le terrain* du code. Cependant, les tests de sécurité exhaustifs ne sont pas forcément réalisables dans ces conditions à cause de l'explosion combinatoire qui en résulte.

Il faudrait tester l'ensemble des scénarii possibles sur l'ensemble du code quelle que soit l'attaque envisagée et ce pour toutes les attaques possibles. Ce travail s'apparente à de la vérification exhaustive d'un programme dans un cadre de sureté. Or, pour la sécurité, on ne s'intéresse qu'aux cas qui peuvent avantager l'attaquant. Le but final étant d'implémenter les contre-mesures appropriées à l'intérieur du code sans sacrifier les performances. Un moyen de vérifier certains aspects de fonctionnement d'un système est d'utiliser le Model Checking ce qui permet de vérifier certains comportement d'un système basé sur une modélisation de celui-ci faisant apparaitre seulement certaines caractéristiques pertinentes vis à vis de la vérification souhaitée.

La problématique associée à cette approche est qu'on est aussi souvent confronté à l'explosion combinatoire mentionnée précédemment. La dernière approche et aussi la plus forte en terme de garanties obtenues est la preuve de programme qui est le dernier domaine de recherche associée à ce travail. La preuve de programme consiste à modéliser à l'aide de formules / d'équations mathématiques le fonctionnement et le comportement d'un programme et de ensuite résoudre ces équations à l'aide d'un solveur afin de prouver que le fonctionnement du programme correspond bien au comportement souhaité.

Chacune de ces approches vient avec un coût en terme d'implémentation et de complexité

qu'il convient de prendre en compte et d'évaluer avant d'envisager son intégration à un processus de développement et un environnement existant.

6.3.2 Cas particulier de la sécurité

Parmi ces domaines, une différence fondamentale existe qui différentie les techniques d'injection d'attaques et la recherche de vulnérabilité de la validation ou vérification de la sécurité.

La différence majeure entre la recherche de vulnérabilités et la validation ou vérification de la sécurité par l'absence de ces vulnérabilités se trouve dans le processus de recherche. Un attaquant se trouve souvent dans la première configuration tandis qu'un défenseur se trouve dans la seconde. Typiquement, un attaquant arrête sa recherche après avoir trouvé la première vulnérabilité tandis que le défenseur doit en théorie toutes les trouver.

6.3.3 Techniques statiques et dynamiques

Un autre critère de différenciation des techniques existantes est l'approche basée sur une analyse statique du code source ou une exécution dynamique de celui-ci. Les approches statiques sont généralement plus faciles à mettre en œuvre mais des résultats précis sont difficiles à obtenir car le contexte d'exécution doit être supposé. Les approches dynamiques sont plus précises en terme de pertinence des résultats fournis mais en fonction du système peuvent être très coûteuses en temps d'exécution. Des solutions hybrides existent mais nécessitent une adaptation au contexte industriel et à la chaine de compilation utilisée. Si celle-ci est modifiée, une nouvelle adaptation doit être effectuée.

6.3.4 Sécurité et absence d'erreurs

Un grand nombre d'attaques exploitent des erreurs présentes dans le code. C'est le cas des débordements de tampon mais ce cas peut être généralisé à la plupart des vérifications nécessaires pour éviter un comportement incontrôlé du programme. Ainsi éliminer les erreurs dans le code peut amener à un renforcement de la sécurité. Vérifier l'absence d'erreurs peut être fait en vérifiant l'absence de certaines fonctions à risque, formes de code sensibles ou bien la présence de la vérification adéquate placée au bon endroit. Dans le cadre des attaques physiques, un raisonnement similaire peut être fait car certaines formes de code peuvent être sensibles vis-à-vis des attaques physiques où l'absence de vérification peut entrainer dans l'hypothèse d'une faute une vulnérabilité.

Différents outils existent permettant ces vérifications.

Les outils propriétaires

Des outils comme Fortify [HP 2012], Klockwork [Klockwork 2012] ou Parasoft [Parasoft 2012] ou Goanna [Fehnker *et al.* 2006] permettent des vérifications du type de celles mentionnées ci-dessus. Cependant, si de tels outils ont fait leurs preuves vis-à-vis de certaines vérifications, leur but reste la détection d'erreur dans le code par la validation, ils ne sont pas spécialisés dans la validation ou la vérification de propriétés de sécurité dans le cadre d'attaques physiques. De plus, dépendre d'une solution propriétaire pour la sécurisation de code peut être rédhibitoire dans certains cadres industriels. Finalement, la configuration de l'outil pour fournir des résultats

pertinents répondant aux questions de sécurité embarqué contre des attaques physiques demande un effort et un maintien coûteux. Ainsi ces outils n'ont pas été retenus bien qu'ils puissent améliorer la qualité du code.

Les outils non propriétaires

Certains outils libres, gratuits ou académiques existent aussi pour la vérification de la sécurité du code source. Parmi ceux-ci, on peut citer :

Splint [Evans & Larochelle 2002] qui fournit par analyse statique des informations sur des potentielles erreurs de codage et l'utilisation de fonctions à risque. Il peut à l'aide d'annotations fournir des informations sur la propagation et la non interférence entre deux variables. Son utilisation s'intègre bien avec des chaines de compilation comme Gcc mais est difficilement adaptable pour passer à l'échelle sur des projets utilisant une autre chaine de compilation.

CppCheck [Marjamäki 2012] vérifie un certain nombre de règles de codage en analysant statiquement le code source. Facile d'utilisation, il n'est cependant pas facilement modifiable pour vérifier d'autres règles.

Sonar [Bellingard *et al.* 2012] vérifie des règles de codage en analysant statiquement le code, il peut facilement être paramétré pour vérifier de nouvelles règles même si ces règles sont limitées aux interfaces fournies par l'outil. Il demande cependant à être intégré à une chaine de compilation spécifique et s'adapte mal aux chaines de compilation propriétaires.

Frama-C [Baudin *et al.* 2012] développé par le CEA, cet outil d'analyse statique par interprétation abstraite permet de vérifier l'absence d'erreurs et aussi de prouver certaines propriétés grâce à un langage d'annotation ACSL. Délicat à prendre en main, il passe cependant très bien à l'échelle grâce à la création abstraite de fonctions de substitution pour le code manquant. Pour effectuer des vérifications sur l'ensemble du code d'un projet embarqué, une adaptation doit cependant être faite et une puissance de calcul conséquente est nécessaire pour mener à bien ces vérifications. Des outils évalués, FramaC semble le plus prometteur bien qu'il n'intègre pas de vérifications liées à des fautes physiques. Un effort d'adaptation a donc été fait pour permettre l'analyse de projets grâce à cet outil qui est utilisé dans le chapitre 4. Bien qu'il soit nécessaire, le coût de la création et du maintien des annotations sur l'ensemble d'un projet pose cependant un problème dans un contexte industriel, ce qui confine son utilisation à un sous-ensemble d'un projet.

Pour une liste exhaustive des outils d'analyse statique existant pour le C, le lecteur peut se référer à [Spinroot 2012]. D'autres outils notamment académiques existent pour la vérification, par exemple d'implémentations cryptographiques, mais l'accès réduit aux sources, les contraintes d'utilisation ou un passage à l'échelle délicat font qu'ils ne sont pas mentionnés ici. Les plates-formes d'injection de fautes ne seront pas non plus couvertes ici car elles agissent principalement sur le binaire résultant de la compilation ce qui ne répond pas à une contrainte industrielle établie précédemment. De plus, celles-ci sont souvent confrontées à des problématiques de passage à l'échelle dans le chapitre 5. Après avoir vérifié si une solution existante ne pouvait pas être utilisée ou dérivée afin de permettre de répondre aux besoins de sécurisation industriel, une solution

spécifique a été conçue et implémentée. Cette solution technique se décompose en l'utilisation de plusieurs outils intervenants à des niveaux différents dans le processus de sécurisation tout en visant à améliorer celui-ci.

6.4 Réalisations techniques

Ces outils se décomposent en deux types d'outils. Les outils d'analyse permettent de comprendre et d'isoler des portions de code critique du point de vue de la sécurité. Les outils de sécurisation permettent d'évaluer ce code vis-à-vis d'attaques physiques.

6.4.1 Amélioration de la relecture de code

Afin d'aider à la relecture de code en cernant le cadre de celle-ci à l'aide de listes de questions servant de rappel, un outil a été sélectionné et modifié afin de pouvoir être utilisé dans le contexte industriel. *Agnitio* [Rook 2011] permet une liaison entre le code source et des besoins en sécurité exprimés sous forme de question. L'utilisation de cet outil permet de cadrer pertinemment le travail du développeur et de l'expert de sécurité en charge de la revue du code source. Les figures 6.2 et 6.1 illustrent l'interface de cet outil.

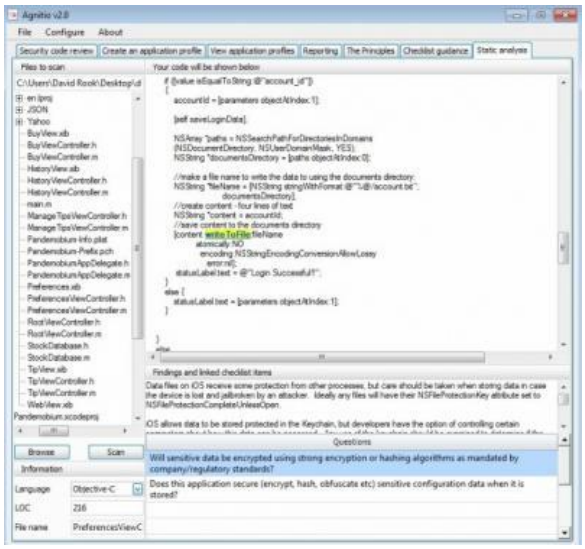


FIGURE 6.1 – Liste de vérification dans le code

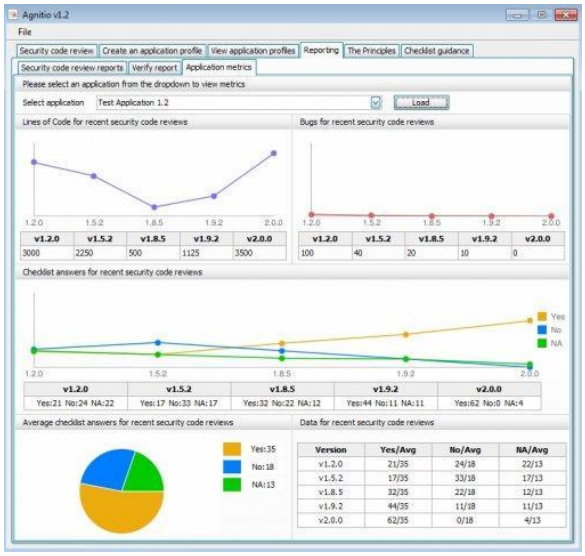


FIGURE 6.2 – Évolution du code au cours des revues

6.4.2 Amélioration de la tracabilité fonctionnelle

Tel que mentionné dans le chapitre 1 section 1.3.7, une approche fonctionnelle est adoptée pour modéliser la sécurité. Cette approche fonctionnelle se justifie par le fait que la sécurité a souvent un sens à ce niveau puisque le gain pour l'attaquant mentionné dans le chapitre 1 section 1.3.7 s'exprime aussi à ce niveau. De plus, l'implémentation de la tâche fonctionnelle est faite dans les sources par le développeur qui est le plus en mesure de comprendre le raisonnement

derrière le code implémenté. Réussir à établir une tracabilité des objectifs fonctionnels jusque dans le code source permet non seulement une meilleure maîtrise du code dans sa réponse aux exigences fonctionnelles mais aussi une meilleure compréhension du code par les autres personnes. Par conséquent un meilleur maintien de celui-ci notamment du point de vue de la sécurité est assuré.

Afin de permettre une documentation du code source et l'ajout d'informations liant une partie du code à sa demande fonctionnelle ou faisant apparaître l'implémentation d'une contre-mesure sécuritaire, le logiciel *Doxygen* [van Heesch 2008] a été utilisé. Ce programme analyse le code source et génère une documentation structurée de celui-ci en faisant apparaître des éléments annotés. Des programmes annexes ont été développés en python pour minimiser l'effort nécessaire pour annoter les éléments requis. Ces outils annexes sont basés sur l'analyse du code source et l'injection de cartouches préremplis contenant des informations extraites du code et requis par *Doxygen*. L'utilisation de ces outils permet une documentation efficace du code source et les documentations générées peuvent être utilisées pour aider à l'identification des points sensibles du code ou à la certification de l'implémentation de sécurité. Les documentations générées ont, de manière pratique, été utilisées dans la justification semi-formelle [Lanet 2000] de telles implémentations. La figure 6.3 illustre l'interface de cet outil.

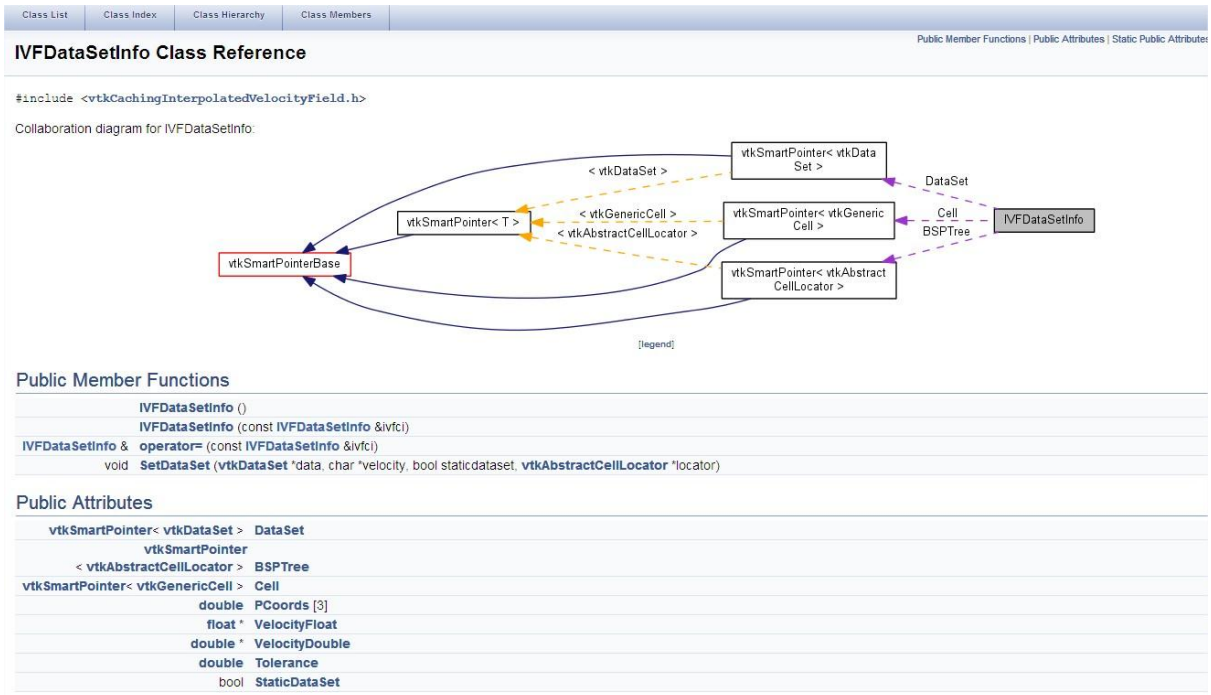


FIGURE 6.3 – Exemple de documentation générée avec Doxygen

6.4.3 Vérification de formes de code

Comme les différents outils mentionnés en section 6.3.4, la vérification de l'absence de certaines formes de code ou de la présence de sécurité peut aider à la sécurisation du code source. D'autres outils basés, sur l'interprétation abstraite et utilisant la plate-forme modulaire de

Listing 6.1– Exemple de code pour catégorisation par mots clés

```
static void foo(int k)
{
  j = pin && pin_cpt || q;
  write_EEPROM(j);
  return j;
}
```

Frama-C, permettent également d'effectuer des vérifications de formes de code ou de flot d'information [Delmas *et al.* 2010] [Cuoq *et al.* 2012]. À l'aide d'une analyse approfondie, ces derniers permettent d'obtenir des résultats sans faux positifs.

Ces outils ont cependant été créés dans une optique de sureté de fonctionnement et non de sécurité. Du point de vue de la sécurité, la recherche et validation de formes de code a pour but d'éviter l'utilisation de constructions vulnérables aux fautes physiques. Afin de valider des règles de codage de sécurité propres à Oberthur Technologies, un *outil d'analyse basé sur la reconnaissance de formes de code* a été développé. Les formes de code sont déduites de l'arbre de syntaxe abstrait (AST) du programme. Cet outil utilise *Pycparser* [Bendersky 2012] pour la création des AST. En se basant sur des formes de code connues, l'outil identifie ces formes ou séquences de formes à l'intérieur de l'intégralité du code source du programme et propose à l'utilisateur une liste de candidats potentiels (extrait de code possédant une vulnérabilité potentielle) en donnant la raison de la sélection du candidat en question.

Des catégories de motifs labellisés sont créés pour guider les analyses. Par exemple, `*pin *::confidentiel` ou `write_*::api_ecriture` pour signifier qu'un élément de code (variable ou fonction) contenant le motif "pin" appartient à la catégorie confidentiel respectivement que tout élément de code commençant par "write_" appartient à la catégorie des interfaces d'écriture. A partir du code du listing 6.1, l'arbre de syntaxe abstrait du listing 6.2 est créé. Les catégories créées sont reflétées sur l'AST et guident ainsi les analyses en déclenchant la recherche d'une nouvelle catégorie ou une analyse à l'endroit de la découverte de la catégorie.

La découverte d'une de ces catégories ou d'une séquence ordonnée de catégories dans l'AST déclenche des vérifications supplémentaires comme la recherche de nouvelles catégories ou des vérifications annexes. Cet outil permet de vérifier un grand nombre de règles de sécurité comme par exemple si des constantes à risque sont utilisées, qu'une forme de code à risque manipulant un objet sensible est utilisée, qu'une réponse sécuritaire se trouve après une écriture en EEPROM ou que des conditions identiques sont utilisées à une distance trop proche (et sont vulnérables à des doubles attaques). La présentation des résultats est réalisée à l'aide de *Sphinx* [Brandl 2012]. La figure 6.4 illustre un exemple de résultats généré par l'outil.

6.4.4 Injection simple et exhaustive d'attaques

Un outil d'injection d'attaque a été développé de manière à pouvoir être intégré au processus de développement existant ainsi qu'à la chaine de compilation. Cet outil devait aussi passer à l'échelle en terme de temps d'exécution des tests. Tout d'abord *une méthode d'injection simple paramétrable* et pouvant être stimulée depuis le terminal à l'aide d'une commande spécifique a été développé. Cet outil utilise une base d'attaques décomposée en motif d'injection et morceau de code stimulant l'attaque. Le développeur peut alors inclure un ou plusieurs motifs d'injec-

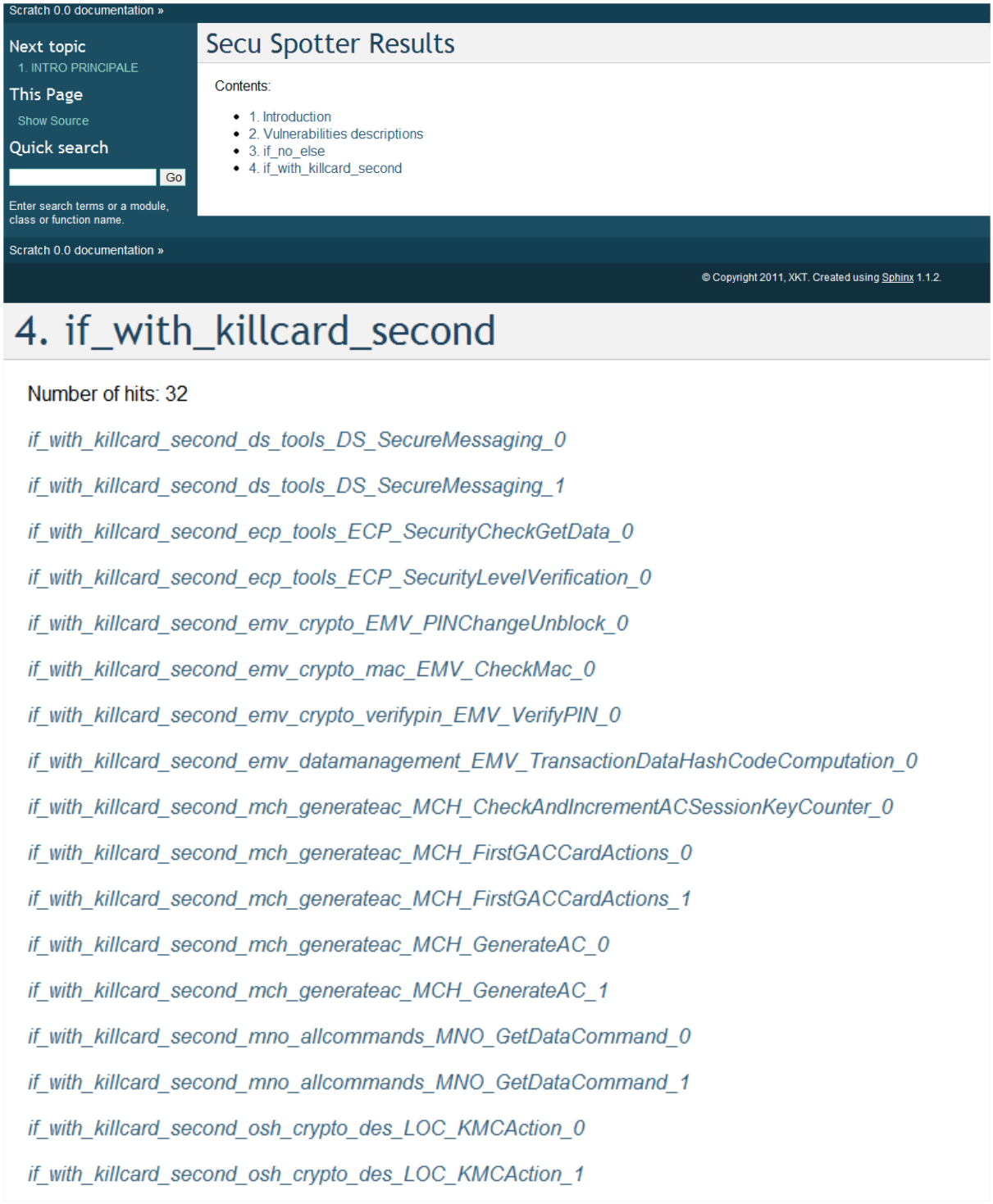


FIGURE 6.4 – Exemple de résultats obtenus par l'outil

Listing 6.2– Exemple de code pour catégorisation par mots clés

```
FileAST:                                                                    1
  FuncDef:                                                                  2
    Decl: foo, [], ['static'], []                                           3
      FuncDecl:                                                             4
        ParamList:                                                          5
          Decl: k, [], [], []                                               6
            TypeDecl: k, []                                                 7
              IdentifierType: ['int']                                       8
        TypeDecl: foo, []                                                   9
          IdentifierType: ['void']                                          10
    Compound:                                                               11
      Assignment: =                                                         12
        ID: j                                                               13
        BinaryOp: ||                                                        14
          BinaryOp: &&                                                      15
            ID: pin                              confidentiel               16
            ID: pin_cpt                          confidentiel               17
          ID: q                                                             18
      FuncCall:                                                             19
        ID: write_EEPROM                         api_ecriture               20
        ExprList:                                                           21
          ID: j                                                             22
      Return: ID: j                                                         23
```

tion dans les commentaires de son code et l'outil les remplace par le morceau de code associé. Un mécanisme utilisant une variable globale de déclenchement est géré afin de pouvoir inclure conditionnellement l'ensemble des morceaux de code d'attaque. Une commande avec une syntaxe spécifique peut ensuite être utilisée pour déclencher au moment de l'exécution du programme une ou plusieurs attaques. En combinant cette méthode avec les contributions du chapitre 4, une campagne de test efficace contre les attaques par modification de valeur peut être créée. Cependant avec cette méthode, les attaques injectées simulent non seulement des modifications de variables à des valeurs spécifiques mais aussi des sauts de code intra-fonction. La méthode nécessite par contre des annotations de la part du développeur. Cette méthode présuppose donc de la part du développeur une idée préalable des attaques susceptibles de mettre en danger son code.

Dans le cas d'un projet inconnu, *Frama-C* peut être utilisé pour analyser le code et déterminer des points d'attaques potentiels. À partir de ces points d'attaque, le développeur peut ensuite positionner dans le code des attaques simulées et observer l'effet de celles-ci. *Frama-C* peut servir à simuler ces attaques en dérivant l'utilisation du mécanisme d'annotation. Il peut cependant aussi servir à vérifier l'appartenance à un ensemble de valeur pour une variable donnée à un point du programme. La figure 6.5 montre l'interface de Frama-C. On peut voir au milieu le code original et à droite une version préprocessée de celui-ci. En se positionnant à un point du programme, par exemple une fonction, on peut, après avoir effectué une analyse de valeur, connaître l'ensemble des variables modifiées par cette fonction avec une profondeur d'appel paramètrable. On peut ainsi savoir si une fonction manipule une variable sensible, ce qui peut guider une attaque. L'interface permet aussi de déterminer les impacts dans le reste du programme d'une instruction et donc d'une modification de celle-ci. La figure 6.6 illustre une telle analyse d'impact. Dans le panneau de gauche, on peut voir l'impact sur les fonctions du programme tandis que le code impacté dans le corps de la fonction est surligné en vert. Les effets

des attaques simulées peuvent être analysées par ces mêmes moyens. En utilisant la méthode développée dans la section 4.3 du chapitre 4, il est possible de déterminer l'ensemble des points d'attaque possibles pour une variable.

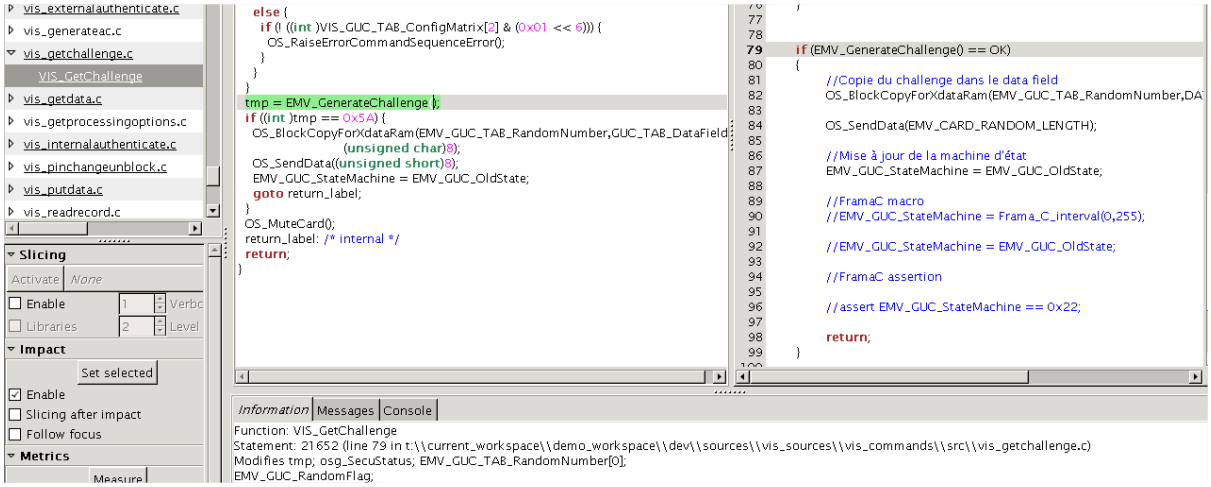


FIGURE 6.5 – Frama-C - Analyse de code

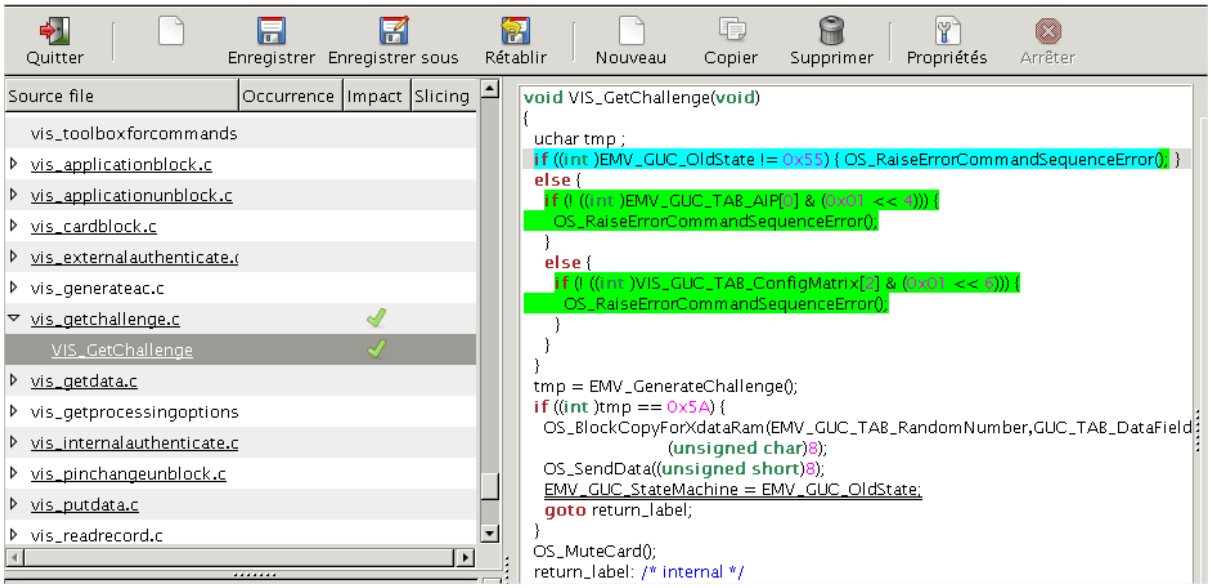


FIGURE 6.6 – Frama-C - Impact d'une instruction

Injecter exhaustivement des attaques permet de s'affranchir de la connaissance préalable de chemins d'attaque et potentiellement d'en découvrir de nouveaux. Dans le cas d'attaques par sauts, *un outil a été développé afin de permettre de simuler exhaustivement l'ensemble des sauts possibles* à l'intérieur d'une fonction. Cet outil basé sur les contributions du chapitre 5, permet de tester sans connaissance préalable un code contre des attaques par saut de code. Cette méthode utilise une instrumentation de chaque instruction d'une fonction avec un couple

de `label` et `goto`. Ces couples sont encapsulés dans des directives de compilation conditionnelle ce qui permet de déclencher un ou plusieurs sauts par exécution. Une variable système est utilisée pour enclencher la compilation d'un `goto` source et d'un `label` de destination. Une fonctionnalité de compilation incrémentale de Visual Studio est utilisée pour ne compiler que le fichier du projet qui est modifié et accélérer les tests. Des optimisations additionnelles sont réalisées à partir d'une exécution préalable du scénario de test choisi qui permet à l'aide de la couverture de code de la fonction d'éliminer les tests inutiles (car le `goto` source ne se trouve pas parmi les lignes de code exécutées). L'ensemble du processus est automatisé à l'aide de scripts. Avec cette méthode, le développeur a seulement à choisir la fonction qu'il veut tester afin de déclencher les campagnes d'attaques. Une campagne de test sur une fonction d'environ 200 LOC prend avec cette méthode entre 5 et 6 heures sur une machine de bureau standard (Dell Latitude D630). En fonction de la chaine de compilation utilisée et des performances de la machine hôte, ces résultats peuvent varier. Une représentation visuelle des résultats catégorisés a été réalisé à l'aide de *Ploticus* [Grubb 2012]. Cette méthode permet de couvrir indirectement un grand nombre d'attaques par valeur modifiant le contrôle de flot du programme. Ainsi l'intégralité des attaques sur des variables impliquées dans des conditions sont testées avec cette méthode d'une manière beaucoup plus efficace qu'en injectant toutes les attaques possibles au niveau assembleur. Des résultats d'une telle campagne sont donnés dans la section 5.5 du chapitre 5.

6.4.5 Analyse et interrogation interactive du code source

Étant donné que la méthode précédente teste exhaustivement toutes les combinaisons de saut possibles à l'intérieur d'une même fonction, le développeur doit quand même spécifier quelle fonction doit être instrumentée pour l'injection d'attaques par saut. Pour simplifier ce processus, un outil a été développé afin d'instrumenter a priori l'intégralité du code avec des instructions permettant de *consigner dans un fichier une trace des fonctions appelées* à l'exécution du scénario choisi. Le développeur peut ainsi utiliser cette trace afin de spécifier les fonctions qu'il veut instrumenter.

Ainsi le développeur peut tester un chemin d'attaque contre l'ensemble des attaques par saut possibles. Cependant, étant donné une fonction ou une opération dans l'intégralité du code d'un projet, il ne peut pas encore connaître l'intégralité des chemins d'attaque permettant d'atteindre ce point. Cet ensemble peut uniquement être obtenu en analysant de manière discrète la totalité des chemins d'appels du programme concernés par cette fonction ou cette opération. Des outils comme PathCrawler [Williams *et al.* 2005] dispose des fonctionnalités nécessaires pour réaliser de telles analyses mais les sources n'étant pas disponibles et afin de garder une cohérence avec le reste des outils développés, une solution spécifique a été développée. L'auteur de [Andouard 2009] utilise une représentation similaire basée sur le graphe d'appel du programme sur lequel des propriétés de logique temporelles peuvent être vérifiées. Ces propriétés temporelles permettent de s'assurer que des séquences d'appels sont bien respectées. L'ergonomie de l'outil en question est plus abouti que celui réalisé dans le cadre de cette thèse. Cependant, les sources n'étant également pas disponibles et afin de garder une cohérence avec le reste des outils développés, une solution alternative accessible par simple ligne de commande a été réalisée. Cet outil se base sur une représentation sous forme de graphe d'appels de l'ensemble des AST de chaque fonction du programme. Il permet un parcours du graphe d'appels tout en disposant d'informations propres à l'AST. Le développeur peut alors à l'aide d'une syntaxe simple, visible dans le listing 6.3

déterminer l'ensemble des chemins d'attaques possibles menant à la fonction `VIS_VerifiyPin` à l'aide d'une simple ligne de commande.

Listing 6.3– Explorateur interactif de code source en ligne de commande

```
cbrowser.py -f VIS_VerifyPin -P -v
# f: fonction
# P: parents recursively in callgraph
# v: create vectors, i.e., all distinctive paths
```

Un fichier de sortie est créé contenant l'ensemble des chemins (un chemin par ligne) sous la forme visible dans le listing 6.4.

Listing 6.4– Exemple de chemins obtenus dans le graphe d'appel du programme

```
VIS_VerifyPIN <- parent-1 <- parent-2a ...
VIS_VerifyPIN <- parent-1 <- parent-2b ...
...
VIS_VerifyPIN <- parent-1a <- parent-2 ..
```

D'autres options permettent interactivement d'obtenir des informations différentes comme l'ensemble des fils et les chemins associés, l'ensemble des chemins d'une fonction à une autre du programme, de filtrer les éléments de sortie en utilisant des expressions régulières, de faire apparaître des éléments de l'AST et même de lancer des analyses comme la recherche d'une séquence particulière parmi les chemins d'attaque ou afficher l'ensemble des variables impliquées dans des conditions sur un chemin d'attaque précis. Il est aussi possible d'obtenir une représentation graphique du graphe d'appels demandé à l'aide de *Graphviz* [Ellson *et al.* 2003] qui est visualisé à l'aide de *ZGRViewer* [Pietriga 2005]. La visualisation supporte différents algorithmes de disposition des nœuds ce qui permet de contrôler la sortie obtenue. Lors de la création des graphes, la possibilité de catégoriser et colorier des groupes de nœuds est aussi offerte. Ces groupes représentent des fonctions dont le nom répond à une expression régulière déterminée. Ainsi, on peut facilement distinguer les fonctions appartenant à un même module si toutes les fonctions de ce module commencent par la même chaine de caractère. Par exemple, "VIS_" pour les fonctions du module VISA. A l'aide de cet outil, un développeur peut améliorer sa connaissance du code mais surtout guider pertinemment des analyses et des tests de son programme d'un point de vue sécuritaire. Grâce à ces fonctionnalités, cet outil est capable de réduire la complexité d'un projet à une échelle compréhensible par le développeur à l'aide de quelques requêtes. Un exemple est fourni avec les figures 6.7 et 6.8 où le programme entier d'une carte à puce est progressivement réduit pour ne faire figurer qu'une partie restreinte du code. Les requêtes permettant d'obtenir ces graphes sont donnés dans le listing 6.5. Par rapport à une simple lecture de code, une telle représentation graphique apporte une vision à haut niveau du système à sécuriser. En raffinant progressivement ses requêtes, le développeur est capable de faire apparaître de manière interactive les seuls éléments intéressants dans son raisonnement. Cette solution offre une méthode d'approche et de recherche alternative aux rapports d'erreurs.

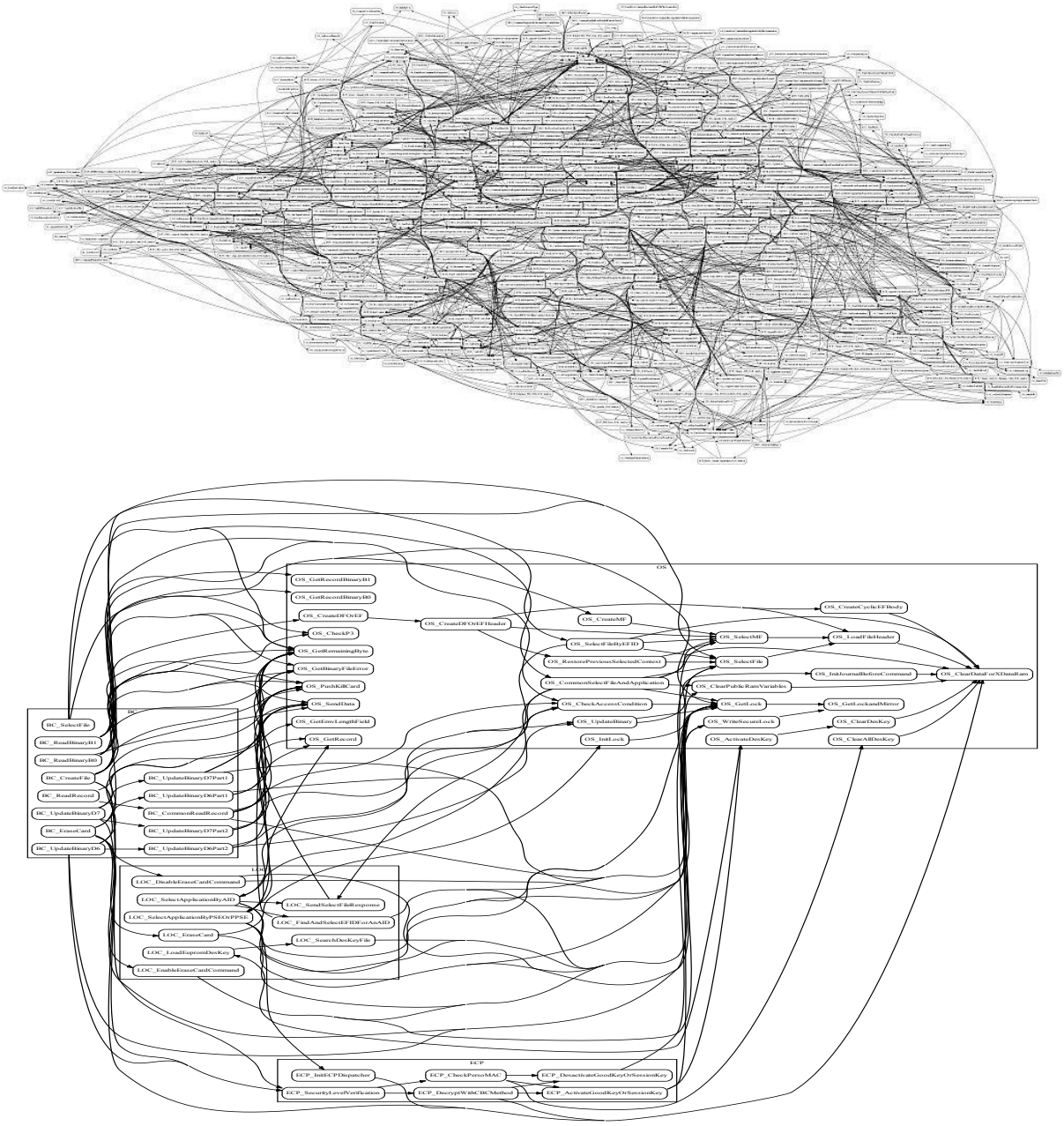
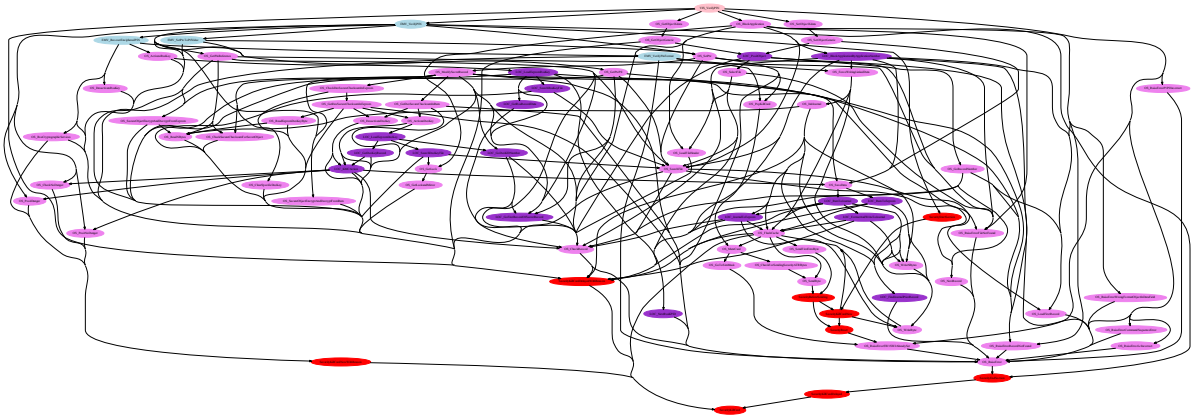
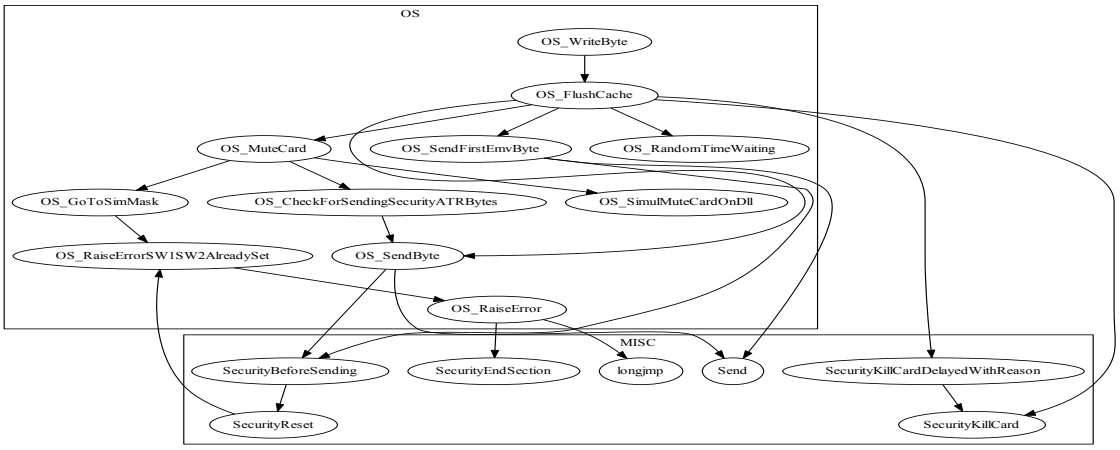


FIGURE 6.7 – Réduction progressive de la complexité du programme par interrogation du graphe d'appels





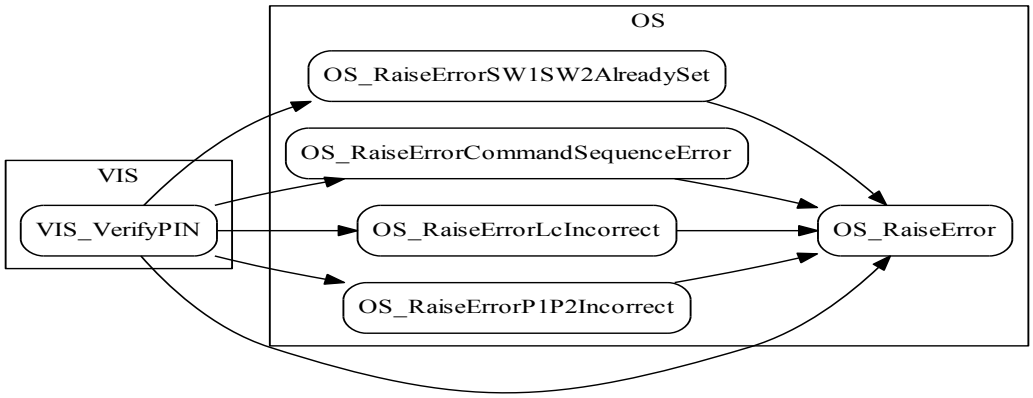


FIGURE 6.8 – Réduction à l'échelle d'un module puis d'un ensemble de fonctions

Listing 6.5– Commandes utilisées pour créer les graphes

```
# détermine l'ensemble des interfaces du module du système d'exploitation
cbrowser.py -f OS_* -p -g

# détermine l'ensemble des interfaces d'une application de la carte au
# système d'exploitation et crée des groupes par module
cbrowser.py -f BC_* -c -d OS_* -G

# détermine l'ensemble des chemins possibles d'une fonction vers une
# autre fonction et colorie les différents modules
cbrowser.py -f VIS_VerifyPIN -c -v -d SecurityKillCard -g --color

# détermine l'ensemble des appels d'une fonction en recréant les chemins
# d'appel permettant de les atteindre et crée des groupes par module
cbrowser.py -f OS_WriteByte -C -v -G

# détermine les chemins possibles d'une fonction à un ensemble d'autres
# fonctions en éliminant certains motifs et crée des groupes
cbrowser.py -f VIS_VerifyPIN -c -v -d OS_Raise* -s [*Transac*, *Command*] -G
```

Finalement, un outil combinant la recherche de formes de code de la section 6.4.3 et l'interrogation interactive du graphe d'appels permet une recherche de séquences complexes de formes de code en s'appuyant sur le graphe d'appels lors de la recherche. Cet outil permet des recherches interprocédurales avec des informations de l'AST. Le listing 6.6 montre un extrait de code en python où un liste définit une séquence de recherche à l'intérieur de l'AST. Avec cette liste, au départ de la fonction `VIS_VerifyPIN`, on cherche une séquence où une sécurité `killcard()` se trouve après une écriture en EEPROM d'une valeur. Dans ce scénario, un attaquant peut fausser la valeur écrite avec une attaque et arracher la carte avant que la défense n'ait eu le temps de se déclencher. Avec les bons paramètres, cette séquence de recherche peut être généralisée à l'ensemble du code source d'un projet.

Listing 6.6– Exemple de recherche de formes de code interprocédurale

```
check_seq = ["FuncDef::Start",
             "+ID(VIS_VerifyPIN)::Down",
             "If@::Down",
             "+Compound::Down",
             "+ID(SecureCounter)::Down",
             "If::Down",
             "+Compound::Down",
             "+ID(killcard)::Up",
             "ID(SecureCounter)?ifcond::Up",
             "@::End"]
check_secu_too_late(check_seq, "FuncCall", "writeEEprom")
```

Appliquée au code source du listing 6.7, cette séquence permet d'obtenir une trace du code entre un `if`, à la ligne 4, qui manipule dans son corps `SecureCounter`, un objet sensible, et un `if`, à la ligne 11, qui vérifie cet objet sensible dans la condition. La fonction `check_secu_too_late` isole les fonctions (`"FuncCall"`) présentes dans la trace et effectue une recherche récursive pour vérifier si un fils dans le graphe d'appels de `f()` ou `g()` n'appelle pas la fonction `writeEEprom`.

Listing 6.7– Exemple de code source pour recherche de formes de code interprocédurale

```
void VIS_VerifyPIN(void)
{
    /* ... */
    if ( /* ... */)             /* début de trace */
    {
        SecureCounter++;
    }
    f();
    if ( /* ... */)
        g();
    if ( SecureCounter != 12 )  /* fin de trace */
        killcard();
    /* ... */
}
```

Disposer à la fois d'informations sur le graphe d'appels et sur l'AST des fonctions au moment de l'analyse permet d'obtenir des informations croisées qui peuvent être utilisées afin d'aider le développeur lors de la sécurisation du code. On peut ainsi créer une table qui donne pour chaque fonction du programme les autres fonctions qui, elles-mêmes ou par un appel fils, utilisent les mêmes variables globales. Un exemple est donné avec la table 6.1. Le développeur peut ainsi, en interrogeant la table, isoler au travers de l'ensemble du programme des fonctions qui peuvent servir de points d'entrée à un attaquant. De tels points d'entrée peuvent servir pour une attaque au travers de la modification de variables globales qui propagent par construction une erreur dans le reste du programme. De telles informations sont difficiles à obtenir sans une analyse interprocédurale du graphe d'appels couplée avec des informations provenant de l'AST.

Fonctions	f1	f2	...	fn
f1	glob1 ;glob6	glob4	...	globi ;globj
f2	glob4	glob4 ;glob7	...	globk ;globl
...	...	...	...	globm ;globn
fn	globi ;globj	globk ;globl	globm ;globn	globx ;globy

TABLE 6.1 – Croisement entre les fonctions et les globales utilisées

Ces analyses se basent sur l'AST de chacune des fonctions du programme analysé. La création des fichiers d'AST prend quelques minutes pour l'ensemble d'un projet. Ces fichiers sont ensuite mis en cache pour les analyses. Ainsi, chacune de ces analyses, même les plus complexes, ne prend que quelques secondes pour s'exécuter, ce qui permet des passes de recherches rapides par un utilisateur. Ces passes peuvent ensuite être regroupées pour créer des suites de recherche de formes de code sensibles sur l'ensemble d'un programme.

6.5 Conclusion

Les méthodologies développées dans les chapitres précédents ont pour but de donner de solides garanties de sécurité au travers d'une formalisation des besoins en sécurité et la modélisation des capacités de l'attaquant. L'application dans un domaine industriel se concrétise par

une amélioration du processus de sécurisation existant à travers l'utilisation d'outils. Ces outils sont en partie basés sur les méthodologies développées dans un contexte académique.

L'utilisation de ces méthodes et outils doit se traduire en un gain en temps et des informations pertinentes remontées aux développeurs chargés de mettre en place la sécurité. Ces outils remplissent les fonctionnalités suivantes : aide à la découverte de nouvelles vulnérabilités, test de résistance des sécurités implémentées contre les attaques connues. Les réalisations dans un contexte industriel incluent :

- Une méthode dédiée à la carte à puce pour tester la sécurité implémentée contre les attaques physiques. Avec cette technique, on peut après annotation du code source, injecter des attaques et les lancer depuis le terminal ;
- Une plate-forme automatique pour le test dynamique des sauts de code et l'identification des impacts fonctionnels. Avec cet outil, il est possible de tester toutes les combinaisons de saut possible dans le code, conséquence d'une attaque physique, et de vérifier leurs impacts fonctionnels.

Le contexte industriel de cette thèse m'a permis de confronter une perspective académique aux contraintes industrielles dans le cadre d'une vérification de la sécurité. Si des outils permettant la preuve de la sécurité sont attractifs du point de vue de la garantie apportée, le coût engendré par leur adoption, leur configuration et leur maintien favorisent à l'heure actuelle des solutions alternatives sur des projets à grande échelle. Je pense cependant que de tels outils dépassent maintenant le cadre académique et peuvent être utilisés avec succès dans un contexte industriel. Il est cependant important de choisir les éléments du système sur lesquels ces outils seront déployés de manière à maximiser le retour sur investissement. Les éléments dont la sécurité est critique et qui ont vocation à être réutilisés figurent parmi les éléments à considérer. Pour le reste du système, des approches qui minimisent l'effort à fournir de la part du développeur et favorisent la génération de résultats se prêtent bien à un contexte industriel. Finalement, je pense que s'il n'existe pas à l'heure actuelle d'outils permettant de garantir de manière automatique, génériquement et sans produire de faux positifs l'ensemble des problématiques de sécurité qui touchent l'embarqué, l'accent doit être mis sur des outils qui augmentent la synergie entre le développeur et l'expert en sécurité. Des outils d'aide à la décision lors de revues de code sont particulièrement appropriés. Je pense aussi qu'il est difficile de vérifier des contraintes de sécurité sans un apport d'information additionnel de la part du développeur. Contrairement à des problématiques de sureté, la sécurité demande au niveau fonctionnel une intention de comportement que le développeur implémente à partir des spécifications. L'objectif lors de la création d'outils d'aide à la décision doit être de réduire la complexité induite par de larges programmes et permettre rapidement et de manière interactive une appréciation de la part de l'utilisateur d'un problème de sécurité.

6.6 Perspectives

Des outils de visualisation de code peuvent être utilisés afin d'aider dans la compréhension du programme et ainsi une sécurisation plus efficace. Les auteurs de [Hurd *et al.* 2010] présentent un outil de visualisation du flot d'information à travers l'intégralité d'un programme C. Cet outil permet notamment de visualiser le passage de l'information de fonctions en fonctions. Les points d'entrée et de sortie de cette information dans une partie du programme est mise en évidence

permettant d'appréhender visuellement la transmission de cette information d'un point à un autre du programme. Des outils similaires ont été développés pour la visualisation à l'exécution [Kurtz 2004] ou des données dans les différentes parties d'un système [Mysore *et al.* 2008]. Les auteurs de [Doernenburg 2006] donnent des exemples d'outils pouvant servir à la création de ces visualisations. Des travaux sur le code assembleur ont également été menées [Ly *et al.* 2008].

Une autre perspective serait d'arriver à mettre en place une technique similaire à la simulation utilisant un débogueur du chapitre précédent. En plus d'offrir de meilleures performances, une solution dynamique permettrait d'interagir directement avec le programme au moment de l'attaque. Des pistes pour de telles implémentations peuvent être trouvées aux références [Nethercote & Seward 2007] et [Brossard 2011].

Chapitre 7

Conclusion

7.1 Conclusion

Les systèmes embarqués sont aujourd'hui utilisés dans de nombreux secteurs d'activités. Parmi ces systèmes, les cartes à puce sont utilisées dans des domaines où la sécurité requiert une attention particulière. Servant de support à de nombreux processus d'authentification, cette carte contient des données sensibles ciblées par des attaquants. Comme le témoigne l'évolution des attaques au cours des 10 dernières années, des attaques de plus en plus sophistiquées sont mises au point permettant d'extraire ces données sensibles mais aussi de perturber le fonctionnement prévu de la carte. Parmi ces attaques, les attaques par faute physique représentent un défi de taille pour les constructeurs de composants et les développeurs logiciels du secteur embarqué. La sécurisation et la certification des plates-formes et du code embarqué représente une phase importante du processus de création d'une carte. Disposer d'outils efficaces permettant d'aider les développeurs confrontés à ce besoin de sécurisation est un enjeu pour les entreprises de développement embarqué. Or, à l'heure actuelle, peu de méthodologies et d'outils prennent en compte cette dimension d'attaque physique. Cette thèse s'inscrit dans ce cadre et a pour but de répondre à ce besoin.

Dans cette thèse, nous avons choisi d'adopter une approche de haut niveau en gardant le point de vue du développeur logiciel. Ainsi, après avoir donné un aperçu de l'architecture interne et du fonctionnement d'une carte à puce, nous avons décrit les différentes attaques existantes. Afin de garantir la sécurité du code contre ces attaques, nous avons exprimé des propriétés de sécurité telles que la confidentialité et l'intégrité au niveau du code source. Exprimer ces propriétés de sécurité à ce niveau a l'avantage de les rendre compréhensibles et utilisables par la personne qui implémente aussi les fonctionnalités logicielles du programme.

Nous avons ensuite cherché à caractériser les capacités d'un attaquant à compromettre le système par une attaque physique. A partir d'un modèle de faute de la littérature, nous avons établi un modèle d'attaque, exprimant au niveau du code source, les conséquences fonctionnelles des attaques. Nous avons également approfondi la caractérisation des effets des fautes physiques, en modélisant les conséquences de celles-ci sur le flot d'interprétation du code par le microprocesseur. Ce modèle nous a ensuite permis de créer des simulations logicielles des différentes attaques possibles. Dans un souci d'optimisation des vérifications, nous avons établi et prouvé une réduction possible du nombre de simulations d'attaque à injecter dans le code source afin de couvrir l'ensemble des points d'attaque.

Une fois ces propriétés de sécurité et un modèle d'attaque exprimés, nous avons cherché à vérifier que les propriétés étaient bien respectées en étudiant le code source du programme. Des éléments observables au niveau du code source, comme le fait qu'une variable ait une valeur particulière à un instant donné du programme, ont été choisis afin de valider le comportement du programme sous attaque. Travailler avec Frama-C, un outil d'analyse statique capable d'effectuer

des analyses sur une représentation abstraite du code, nous a permis d'étendre ce critère à l'appartenance d'une valeur d'une variable à un intervalle de valeur à un instant donné de l'exécution. En combinant des éléments observables, validant le comportement du code, à des attaques physiques, simulées par injection de code source, nous avons proposé une méthodologie permettant de valider statiquement la sécurité du comportement du code sous attaque.

Face à certaines limitations de l'analyse statique, une méthodologie, basée sur le test et la validation dynamique d'éléments observables, à été mise au point. Cette méthode utilise un principe d'injection de code issu du modèle d'attaque simulant des attaques par saut. Nous avons utilisé cette méthode expérimentale afin de découvrir et d'isoler de nouvelles attaques correspondant au modèle d'attaque. Une représentation graphique de la répartition des attaques potentiellement dangereuses a été proposée afin de visualiser, sur l'ensemble du programme, les zones à risque à l'échelle d'une fonction. Cette méthode nous a également servi à confronter la couverture de notre modèle d'attaque à haut niveau avec la totalité des attaques possibles en assembleur. Nous avons pour cela utilisé deux techniques d'injections : avant compilation et à la volée pendant l'exécution dynamique du programme à l'aide d'un débogueur. Ces techniques intervenant au niveau C ou au niveau assembleur. Ces différentes techniques exposent ainsi différents moyens de mise en œuvre possibles. La méthodologie a finalement été portée et testée dans un environnement industriel où elle a mise en évidence une vulnérabilité sur le code de carte à puce. L'attaque associée a cependant été jugée très difficile voire impossible à réaliser avec les compétences actuelles des attaquants.

Une dernière partie de la thèse traite des différentes méthodes et outils développés dans le contexte industriel, ayant pour but d'améliorer le processus de sécurisation du code source. Ces méthodes incluent le test et le passage à l'échelle d'outils d'analyse statique sur des projets industriels. Une solution dédiée, composée d'outils d'analyse de code, de parcours et d'interrogation du graphe d'appels, a également été développée afin de faciliter le travail du développeur chargé de sécuriser le code source. Cette partie donne un exemple d'informations dont celui-ci doit disposer afin de sécuriser efficacement son code source.

Pour conclure, on peut faire appel à la figure 1.15 du chapitre 1 qui donne une vue d'ensemble de la problématique de sécurité dans un contexte embarqué. Cette figure montre que la sécurité fait intervenir des aspects matériel, fonctionnel et d'attaque. Cette thèse montre qu'un aspect de modélisation important entre en jeu. Cette modélisation est nécessaire afin d'exprimer efficacement des besoins en sécurité, cerner l'ensemble des possibilités d'attaque et élaborer des solutions appropriées. Si cette thèse propose une solution purement logicielle au problème de la sécurité embarqué, cette approche ne doit pas faire oublier qu'il est important de considérer conjointement les perspectives matérielles et logicielles afin de cerner efficacement l'ensemble de la complexité du problème [Wolf 1994].

7.2 Perspectives

Les résultats obtenus dans cette thèse peuvent être étendus suivant plusieurs axes. Ces axes ont trait à des domaines théoriques et pratiques de l'informatique. Ces axes incluent :

La sécurité où des modélisations adoptant des perspectives différentes permettraient de rationaliser des problématiques complexes ;

L'architecture sécurisée avec une quantification de la sécurité par rapport à l'ensemble des défenses matérielles permettant de déduire les besoins logiciels ;

La compilation avec la création de compilateurs certifiés garantissant des propriétés de sécurité ou aidant à la vérification de celles-ci ;

Le langage où la sémantique et l'interprétation abstraite appliqués à la sécurité ouvrent de nouvelles perspectives d'analyse ;

L'ingénierie logicielle avec la création de tests et d'injections de fautes dédiées à la sécurité ;

La théorie de l'information qui permettrait de raffiner les modèles établis ;

La visualisation qui appliquée à la sécurité permettrait de trouver des représentations adaptées.

Les travaux réalisés dans cette thèse ouvrent des perspectives d'intégration avec d'autres domaines mais aussi des perspectives d'amélioration et de raffinement du travail existant. Par exemple, intégrer une perspective d'attaque physique à des outils d'analyse statique ou dynamique existants, en se basant sur les modélisations établies, pourrait permettre d'élaborer des analyses sophistiquées exploitant les capacités de ces outils. Ces analyses auraient pour but, soit d'aider le développeur dans son processus de sécurisation, en mettant à sa disposition des informations supplémentaires, soit de tester le code et les sécurités implémentées pour évaluer leur résistance à des attaques physiques modélisées.

Les propriétés de sécurité et modèles établis pourraient également être étendus afin de faire apparaître des éléments de théorie de l'information ou des aspects de séquencement propre à la logique temporelle. Ces propriétés, une fois établies sur l'ensemble du système, constituent une politique de sécurité qui assure un niveau de sécurité à travers l'ensemble du système. Cependant, les politiques générées peuvent contenir des conflits qu'il serait intéressant d'étudier et de résoudre afin d'obtenir une cohérence dans la sécurité sur l'ensemble du système.

Les tests de sécurité établis ici pourraient être améliorés en se rapprochant d'autres domaines comme le *concolic testing* ou le *fuzzing*. Ces domaines proposent des techniques de découverte et génération automatique de vecteurs d'entrée permettant d'atteindre un point particulier du programme. Établir automatiquement des tests pertinents pourraient représenter un réel avantage dans un contexte industriel. Des techniques comme la couverture de code pourraient également être utilisées pour améliorer la visibilité des tests établis et permettre des comparaisons de traces d'exécution. Dans une même optique, des visualisations plus sophistiquées pourraient être établies pour mettre en évidence des aspects de sécurité du code à l'échelle d'un projet.

Prendre en compte les optimisations du compilateur du point de vue de la sécurité représente également un défi de taille. Réussir à traduire des propriétés de haut niveau se répercutant sur le code assembleur ou une modélisation de bas niveau pourrait être une piste de travaux futurs. Dans un contexte un peu différent, où le compilateur peut être modifié, celui-ci pourrait être utilisé afin d'extraire des informations sur le code au moment de la compilation où même de sécuriser automatiquement des portions de code. Cette solution aurait l'avantage de déléguer la sécurisation et simplifier le travail de sécurisation du développeur. Sans l'aide du compilateur, une instrumentation du code pour la sécurité peut également être effectuée. Un exemple d'une telle instrumentation est donné dans cette thèse.

Des perspectives propres à chaque chapitre sont aussi présentées à la fin de ceux-ci.

Bibliographie

[Agostini 2009] O. Agostini. Simulateur d'Attaques Hardware sur Carte à Puce. Master's thesis, Université de Bordeaux 1, 2009. (Cité en page 145.)

[Agrawal *et al.* 2002] D. Agrawal, B. Archambeault, J. Rao et P. Rohatgi. *The EM Side-Channel(s).* In B. Kaliski, Ç. K. Koç et C. Paar, editeurs, Cryptographic Hardware and Embedded Systems - CHES, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer Berlin / Heidelberg, 2002. (Cité en page 83.)

[Aigner & Oswald 2011] M. Aigner et E. Oswald. *Power Analysis Tutorial.* Rapport technique, University of Technology Graz, 2011. (Cité en page 84.)

[Akkar *et al.* 2000] M.-L. Akkar, R. Bevan, P. Dischamp et D. Moyart. *Power Analysis, What Is Now Possible...* In T. Okamoto, editeur, ASIACRYPT, volume 1976 of *Lecture Notes in Computer Science*, pages 489–502. Springer, 2000. (Cité en page 83.)

[Akkar *et al.* 2003] M.-L. Akkar, L. Goubin et O. Ly. *Automatic Integration of Counter-Measures Against Fault Injection Attacks.* In e-Smart, Nice, France, 2003. (Cité en page 167.)

[Amiel *et al.* 2007] F. Amiel, K. Villegas, B. Feix et L. Marcel. *Passive and Active Combined Attacks : Combining Fault Attacks and Side Channel Analysis.* In Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 92–102, Vienna, Austria, 2007. IEEE. (Cité en page 84.)

[Anderson & Kuhn 1996] R. Anderson et M. Kuhn. *Tamper resistance-a cautionary note.* In Second USENIX Workshop on Electronic Commerce, volume 2, pages 1 – 11, San Diego, CA, USA, 1996. (Cité en page 27.)

[Anderson & Kuhn 1997] R. Anderson et M. Kuhn. *Low Cost Attacks on Tamper Resistant Devices.* In 5th International Workshop Security Protocols, pages 125–136, Paris, France, 1997. Springer-Verlag. (Cité en pages 85 et 86.)

[Andouard 2009] P. Andouard. *Outils d'aide à la recherche de vulnérabilités dans l'implantation d'applications embarquées sur carte à puce.* PhD thesis, Université de Bordeaux 1, 2009. (Cité en pages 77, 145 et 181.)

[Andronick *et al.* 2005] J. Andronick, B. Chetali et C. Paulin-Mohring. *Formal Verification of Security Properties of Smart Card Embedded Source Code.* In J. A. Fitzgerald, I. J. Hayes et A. Tarlecki, editeurs, Formal Methods - FM, volume 3582 of *Lecture Notes in Computer Science*, pages 302–317, Newcastle, UK, 2005. Springer. (Cité en page 51.)

[Angelis 2012] D. Angelis. *Motorola 68000 Instruction Set*, 2012. http://infoindustrielle.free.fr/68K_pdf/02_68K_36_106.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 39.)

[Aumüller *et al.* 2002] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter et J.-P. Seifert. *Fault Attacks on RSA with CRT : Concrete Results and Practical Countermeasures.* In B. S. Kaliski Jr., Ç. K. Koç et C. Paar, editeurs, Cryptographic Hardware and Embedded Systems - CHES, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275, Redwood Shores, CA, USA, 2002. Springer. (Cité en page 85.)

[Avenel 2010] Y. Avenel. *Cartes à puces : Les nouvelles frontières*, 2010. http://www.electroniques.biz/pdf/ES_2010_010_044.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 3.)

[Balliu & Mastroeni 2009] M. Balliu et I. Mastroeni. *A weakest precondition approach to active attacks analysis.* In S. Chong et D. A. Naumann, editeurs, Workshop on Programming Languages and Analysis for Security - PLAS, pages 59–71. ACM, 2009. (Cité en pages 76 et 77.)

[Bar-El *et al.* 2006] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall et C. Whelan. *The sorcerer's apprentice guide to fault attacks.* Proceedings of the IEEE, vol. 94, no. 2, pages 370–382, 2006. (Cité en page 85.)

[Bar-El 2003] H. Bar-El. *Known Attacks Against Smartcards.* Rapport technique, Discretix Technologies, 2003. http://www.hbarel.com/publications/Known_Attacks_Against_Smartcards.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 29.)

[Barbosa *et al.* 2005] R. Barbosa, J. Vinter, P. Folkesson et J. Karlsson. *Assembly-Level pre-injection analysis for improving fault injection efficiency.* In 5th European conference on Dependable Computing - EDCC, EDCC'05, pages 246–262, Budapest, Hungary, 2005. Springer-Verlag. (Cité en page 119.)

[Barbu *et al.* 2010] G. Barbu, H. Thiebeauld et V. Guerin. *Attacks on Java Card 3.0 Combining Fault and Logical Attacks.* In J. I.-C. Dieter Gollmann Jean-Louis Lanet, editeur, Smart Card Research and Advanced Application. 9th IFIP WG 8.8/11.2 International Conference, volume 6035 of *Lecture Notes in Computer Science / Security & Cryptology*, pages 148–163, Passau, Allemagne, 2010. Springer. (Cité en page 44.)

[Baudin *et al.* 2010] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy et V. Prevosto. *ACSL : ANSI/ISO C Specification Language.* Rapport technique, CEA LIST and INRIA, 2009-2010. Preliminary Design (v 1.4). (Cité en pages 111 et 113.)

[Baudin *et al.* 2012] P. Baudin, R. Bonichon, L. Correnson, P. Cuoq, Z. Dargaye, J.-C. Filliâtre, P. Herrmann, C. Marché, B. Monate, Y. Moy, A. Pacalet, V. Prévosto, J. Signoles et B. Yakobowski. *Frama-C : Framework for Modular Analysis of C.* CEA-LIST and INRIA-Futurs, 2012. http://frama-c.com/ [En ligne ; consulté le 05-août-2012]. (Cité en page 174.)

[Bayart 2012] F. Bayart. *La sécurité des cartes bancaires*, 2012. http://www.bibmath.net/crypto/moderne/cb.php3 [En ligne ; consulté le 05-août-2012]. (Cité en pages 7 et 211.)

[Bell & LaPadula 1975] D. E. Bell et L. J. LaPadula. *Secure Computer Systems : Unified Exposition and Multics Interpretation.* Rapport technique MTR-2997, The MITRE Corp., 1975. (Cité en page 50.)

[Bellingard *et al.* 2012] F. Bellingard, D. Bolkensteyn, S. Brandhof, O. Gaudin, F. Mallet et E. Mandrikov. *Sonar an open platform to manage code quality*, 2012. http://www.sonarsource.org/ [En ligne ; consulté le 05-août-2012]. (Cité en page 174.)

[Bendersky 2012] E. Bendersky. *Pycparser tool for parsing C code*, 2012. http://code.google.com/p/pycparser/ [En ligne ; consulté le 05-août-2012]. (Cité en page 177.)

[Berthomé *et al.* 2010] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. *Attack model for verification of interval security properties for smart card C codes.* In 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security PLAS'10, pages 1–12, Toronto Canada, 2010. ACM. (Cité en pages 46 et 110.)

[Berthomé *et al.* 2011] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. *Simulating physical attacks in smart card C codes : the jump attack case.* In e-Smart, Nice, France, 2011. (Cité en pages 46 et 169.)

[Berthomé *et al.* 2012] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. *High level model of control flow attacks for smart card functional security.* In 7th International Conference on Availability, Reliability and Security AReS 2012, Prague, Czech Republic, 2012. IEEE Computer Society. (Cité en pages 46 et 143.)

[Biba 1977] K. J. Biba. *Integrity Considerations for Secure Computer Systems.* Rapport technique, MITRE Corp., 1977. (Cité en page 50.)

[Biham & Shamir 1997] E. Biham et A. Shamir. *Differential Fault Analysis of Secret Key Cryptosystems.* In Advances in Cryptology - CRYPTO, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525, Santa Barbara, California, USA, 1997. Springer Berlin / Heidelberg. (Cité en page 85.)

[Blazy & Leroy 2005] S. Blazy et X. Leroy. *Formal verification of a memory model for C-like imperative languages.* In R. B. Kung-Kiu Lau, editeur, ICFEM'05 : 7th International Conference on Formal Engineering Methods, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299, Manchester, UK, 2005. Springer. http ://www.springer.com/. (Cité en page 52.)

[Blazy *et al.* 2006] S. Blazy, Z. Dargaye et X. Leroy. *Formal Verification of a C Compiler Front-End.* In International Symposium on Formal Methods - FM, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, Hamilton, ON, Canada, 2006. Springer. (Cité en page 79.)

[Blömer *et al.* 2006] J. Blömer, M. Otto et J.-P. Seifert. *Sign Change Fault Attacks on Elliptic Curve Cryptosystems.* In L. Breveglieri, I. Koren, D. Naccache et J.-P. Seifert, editeurs, Fault Diagnosis and Tolerance in Cryptography, volume 4236 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2006. (Cité en page 93.)

[Bodescot & Guinot 2011] A. Bodescot et D. Guinot. *Les cartes à puce "sans contact" se généralisent.* Le Figaro, 2011. http://www.eurosmart.com/images/doc/articles/111114-lefigaro.fr-itwm.bertin.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 3.)

[Bond & Anderson 2001] M. Bond et R. J. Anderson. *API-Level Attacks on Embedded Systems.* IEEE Computer, vol. 34, no. 10, pages 67–75, 2001. (Cité en page 18.)

[Bond 2006] M. Bond. *Phish and Chips.* In B. Christianson, B. Crispo, J. A. Malcolm et M. Roe, editeurs, Security Protocols Workshop, volume 5087 of *Lecture Notes in Computer Science*, pages 49–51, Cambridge, UK, 2006. Springer. (Cité en page 16.)

[Boneh *et al.* 2001] D. Boneh, R. DeMillo et R. Lipton. *On the importance of checking cryptographic protocols for faults.* Journal of Cryptology, vol. 14, no. 2, pages 101–119, 2001. (Cité en page 85.)

[Bouzefrane 2008] S. Bouzefrane. *Cours carte à puce EMV*, 2008. http://cedric.cnam.fr/~bouzefra/cours/cours_SEM/Cartes_Bouzefrane_EMV.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 18.)

[Bouzefrane 2009] S. Bouzefrane. *Cours carte à puce*, 2009. http://cedric.cnam.fr/~bouzefra/cours/cours_SEM/Cartes_Bouzefrane_partie1.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 10.)

[Brandl 2012] G. Brandl. *Sphinx Python Documentation Generator*, 2012. http://sphinx.pocoo.org/ [En ligne ; consulté le 05-août-2012]. (Cité en page 177.)

[Brossard 2011] J. Brossard. *Post Memory CorruptionMemory Analysis.* In Black Hat, Las Vegas, NV, USA, 2011. http://www.pmcma.org/wp-content/uploads/2011/09/BHUS-2011_Brossard.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 188.)

[Buetler 2008] I. Buetler. *Smart Card APDU Analysis.* In Black Hat, Las Vegas, NV, USA, 2008. http://www.blackhat.com/presentations/bh-usa-08/Buetler/BH_US_08_Buetler_SmartCard_APDU_Analysis_V1_0_2.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 18.)

[Canivet 2009] G. Canivet. *Analyse des effets d'attaques par fautes et conception sécurisée sur plate-forme reconfigurable.* PhD thesis, Institut Polytechnique de Grenoble, 2009. (Cité en page 86.)

[CCP 2012] *Common Criteria Portal*, 2012. www.commoncriteriaportal.com [En ligne ; consulté le 05-août-2012]. (Cité en page 11.)

[Chambérot & Kauffmann-Tourkestansky 2009] F. Chambérot et X. Kauffmann-Tourkestansky. *Dedicated Smart Card Security Checking.* In e-Smart, Nice, France, 2009. (Cité en pages 46 et 169.)

[Christmansson & Chillarege 1996] J. Christmansson et R. Chillarege. *Generation of an error set that emulates software faults based on field data.* In Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96), FTCS '96, pages 304–, Washington, DC, USA, 1996. IEEE Computer Society. (Cité en page 144.)

[Cifuentes & Gough 1995] C. Cifuentes et K. J. Gough. *Decompilation of Binary Programs.* Software - Practice & Experience, vol. 25, no. 7, pages 811–829, 1995. (Cité en pages 92, 93 et 139.)

[Clavier *et al.* 2010] C. Clavier, B. Feix, G. Gagnerot et M. Roussellet. *Passive and Active Combined Attacks on AES - Combining Fault Attacks and Side Channel Analysis.* In L. Breveglieri, M. Joye, I. Koren, D. Naccache et I. Verbauwhede, editeurs, Fault Diagnosis and Tolerance in Cryptography, pages 10–19, Santa Barbara, CA, USA, 2010. IEEE Computer Society. (Cité en page 84.)

[Clavier 2007] C. Clavier. *De la sécurité physique des crypto-systèmes embarqués.* PhD thesis, Université de Versailles, St Quentin en Yvelines, 2007. (Cité en page 86.)

[Collin 2005] L. Collin. *A Quick Benchmark : Gzip vs. Bzip2 vs. LZMA*, 2005. http://tukaani.org/lzma/benchmarks.html [En ligne ; consulté le 05-août-2012]. (Cité en page 155.)

[Consultants 2009] D. N. Consultants. *Dimension économique et industrielle des cartes à puces*, 2009. http://www.industrie.gouv.fr/p3e/etudes/cartes_puces/cartes_puces.pdf [En ligne ; consulté le 05-août-2012]. (Cité en pages 3, 20 et 211.)

[Coron *et al.* 2007] J.-S. Coron, E. Prouff et M. Rivain. *Side Channel Cryptanalysis of a Higher Order Masking Scheme.* In P. Paillier et I. Verbauwhede, editeurs, Cryptographic Hardware and Embedded Systems - CHES, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007. (Cité en page 78.)

[Correnson *et al.* 2010] L. Correnson, P. Cuoq, A. Puccetti et J. Signoles. *Frama-C User Manual.* CEA LIST, 2010. (Cité en page 111.)

[Cortier 2005] V. Cortier. *Verification of cryptographic protocols : techniques , tools and link to cryptanalysis.* In Symposium A Quarterly Journal In Modern Foreign Literatures, Nancy, France, 2005. (Cité en page 28.)

[Cousot & Cousot 1977] P. Cousot et R. Cousot. *Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In R. M. Graham, M. A. Harrison et R. Sethi, editeurs, Symposium on Principles of Programming Languages - POPL, pages 238–252, Los Angeles, CA, USA, 1977. ACM. (Cité en page 51.)

[Criteria 2009] C. Criteria. *Application of Attack Potential to Smartcards.* Rapport technique March, BSI, 2009. www.commoncriteriaportal.org/files/supdocs/CCDB-2009-03-001.pdf [En ligne ; consulté le 05-août-2012]. (Cité en pages 10, 21 et 90.)

[Cuoq & Prevosto 2010] P. Cuoq et V. Prevosto. *Frama-C's value analysis plug-in.* Rapport technique, CEA LIST, 2010. http://frama-c.com/download/value-analysis-Boron-20100401.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 114.)

[Cuoq *et al.* 2012] P. Cuoq, D. Delmas, S. Duprat et V. M. Lamiel. *Fan-C, a Frama-C plug-in for data flow verification.* In Embedded Real Time Software and Systems ERTS, Toulouse , France, 2012. (Cité en page 177.)

[De Haas 2007] J. De Haas. *Side Channel Attacks and Countermeasures for Embedded Systems.* In Black Hat, Las Vegas, NV, USA, 2007. (Cité en page 82.)

[Delmas *et al.* 2010] D. Delmas, S. Duprat, V. M. Lamiel et J. Signoles. *Taster, a Frama-C plug-in to enforce Coding Standards.* In Embedded Real Time Software and Systems ERTS, Toulouse , France, 2010. (Cité en page 177.)

[Denning 1976] D. E. Denning. *A Lattice Model of Secure Information Flow.* Papers from Fifth ACM Symposium on Operating Systems Principles, vol. 19, no. 5, pages 236–243, 1976. (Cité en page 51.)

[Derouet 2007] O. Derouet. *Secure Smartcard Design against Laser Fault Injection.* In 4th Workshop on Fault Diagnostic and Tolerance in Cryptography, Vienne, Autriche, 2007. (Cité en page 87.)

[Descamps 2000] J. B. N. Descamps. La carte à puce. Master's thesis, Université de Lille 1, 2000. http://phonecards.free.fr/these.htm [En ligne ; consulté le 05-août-2012]. (Cité en page 8.)

[Doernenburg 2006] E. Doernenburg. *Software Visualization and Model Generation.* Engineer, pages 1–12, 2006. (Cité en page 188.)

[Dutertre *et al.* 2009] J.-M. Dutertre, A. Tria, M. Agoyan, B. Robisson et M. Agoyan. *Low cost fault injection method for security characterization Secure ICs design issues.* In e-Smart, Nice, France, 2009. (Cité en page 13.)

[Dutertre *et al.* 2010] J.-M. Dutertre, A.-P. Mirbaha, A. Tria, B. Robisson et M. Agoyan. *Revue expérimentale des techniques d'injection de fautes.* Journée Sécurité – GDR SoC-SiP, 2010. (Cité en page 86.)

[Ellson *et al.* 2003] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North et G. Woodhull. *Graphviz and dynagraph - static and dynamic graph drawing tools.* In Graph Drawing Software, pages 127–148. Springer-Verlag, 2003. (Cité en page 182.)

[Eurosmart 2001] Eurosmart. *Common Criteria for IT Security Evaluation Protection Profile – Smartcard Integrated Circuit Protection Profile*, 2001. (Cité en page 10.)

[Evans & Larochelle 2002] D. Evans et D. Larochelle. *Improving Security Using Extensible Lightweight Static Analysis.* IEEE Software, vol. 19, no. 1, pages 42–51, 2002. http://www.splint.org [En ligne ; consulté le 05-août-2012]. (Cité en page 174.)

[EventHelix 2012] EventHelix. *C to Assembly Translation*, 2012. http://www.eventhelix.com/RealtimeMantra/Basics/CToAssemblyTranslation.htm [En ligne ; consulté le 05-août-2012]. (Cité en page 38.)

[Faurax 2008] O. Faurax. *Évaluation par simulation de la sécurité des circuits face aux attaques par faute.* These, Université de la Méditerranée - Aix-Marseille II, 2008. (Cité en page 145.)

[Fehnker *et al.* 2006] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg et F. Rauch. *Goanna - A Static Model Checker.* In L. Brim, B. R. Haverkort, M. Leucker et J. van de Pol, editeurs, Formal Methods : Applications and Technology, volume 4346 of *Lecture Notes in Computer Science*, pages 297–300. Springer, 2006. http://redlizards.com/products/ [En ligne ; consulté le 05-août-2012]. (Cité en page 173.)

[Flottes *et al.* 2011] M. L. Flottes, G. D. Natale et G. Gogniat. *Journée sécurité : Sécurité des systèmes embarqués*, 2011. http://www2.lirmm.fr/journees_securite/material/prospectives/2011.pdf [En ligne ; consulté le 05-août-2012]. (Cité en page 11.)

[Gadellaa 2005] K. O. Gadellaa. *Fault Attacks on Java Card.* PhD thesis, Technische Universiteit Eindhoven, 2005. (Cité en pages 85 et 86.)

[Gandolfi *et al.* 2001] K. Gandolfi, C. Mourtel et F. Olivier. *Electromagnetic Analysis : Concrete Results.* In Ç. Koç, D. Naccache et C. Paar, editeurs, Cryptographic Hardware and Embedded Systems - CHES, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer Berlin / Heidelberg, 2001. (Cité en page 83.)

[Giraud & Thiebeauld 2004a] C. Giraud et H. Thiebeauld. *Basics of Fault Attacks.* In Fault Diagnosis and Tolerance in Cryptography, Florence, Italy, 2004. (Cité en page 85.)

[Giraud & Thiebeauld 2004b] C. Giraud et H. Thiebeauld. *A Survey on Fault Attacks.* In J.-J. Quisquater, P. Paradinas, Y. Deswarte et A. A. E. Kalam, editeurs, Conference on Smart Card Research and Advanced Applications - CARDIS, pages 159–176, Toulouse, France, 2004. Kluwer. (Cité en page 85.)

[Giraud 2007] C. Giraud. *Attaques de cryptosystèmes embarqués et contre-mesures associées.* PhD thesis, Université de Versailles, St Quentin en Yvelines, 2007. (Cité en page 84.)

[Gobio *et al.* 2000] H. Gobio, S. Smith, J. D. Tygar et B. Yee. *Smart Cards in Hostile Environments.* Security, 2000. (Cité en page 28.)

[Goguen & Meseguer 1982] J. A. Goguen et J. Meseguer. *Security Policies and Security Models.* In IEEE Symposium on Security and Privacy, pages 11–20, Oakland, CA, USA, 1982. (Cité en page 51.)

[Goldwasser *et al.* 1989] S. Goldwasser, S. Micali et C. Rackoff. *The knowledge complexity of interactive proof systems.* SIAM Journal on Computing, vol. 18, no. 1, pages 186–208, 1989. (Cité en page 4.)

[Griffith & Kaiser 2007] R. Griffith et A. G. E. Kaiser. *Evaluating Software Systems via Fault-Injection and Reliability, Availability and Serviceability (RAS) Metrics and Models Thesis proposal.* Science, 2007. (Cité en page 93.)

[Grinschgl *et al.* 2011] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock et J. Haid. *Automatic saboteur placement for emulation-based multi-bit fault injection.* In 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip - ReCoSoC, pages 1–8, Montpellier, France, 2011. IEEE. (Cité en page 144.)

[Grubb 2012] S. Grubb. *Software package for producing plots, charts and graphics from data*, 2012. http://ploticus.sourceforge.net/doc/welcome.html [En ligne ; consulté le 05-août-2012]. (Cité en page 181.)

[Guilley *et al.* 2008] S. Guilley, L. Sauvage, J.-L. Danger, N. Selmane et R. Pacalet. *Silicon-level Solutions to Counteract Passive and Active Attacks.* In IEEE-CS, editeur, 5th workshop on Fault Tolerance and Detection in Cryptography, pages 3–17, Washington, DC, USA, 2008. IEEE-CS. (Cité en page 27.)

[Guilley *et al.* 2010] S. Guilley, L. Sauvage, J.-L. Danger et N. Selmane. *Fault Injection Resilience.* In Fault Diagnosis and Tolerance in Cryptography, pages 51–65, Santa Barbara, United States, 2010. IEEE Computer Society. 17 pages (extended version). (Cité en page 27.)

[Guilley 2007] S. Guilley. *Architecture et CAO pour crypto-processeurs sécurisés.* In GDR SoC-SiP, 2007. (Cité en pages 26, 86 et 211.)

[Henning 2006] J. L. Henning. *SPEC CPU2006 benchmark descriptions.* ACM SIGARCH Computer Architecture News, vol. 34, no. 4, 2006. (Cité en page 150.)

[Hotz 2010] G. Hotz. *PS3 Hack Pastie*, 2010. http://pastie.org/795944 [En ligne ; consulté le 05-août-2012]. (Cité en page 1.)

[HP 2012] HP. *HP Fortify Static Code Analyzer (SCA)*, 2012. https://www.fortify.com/products/hpfssc/source-code-analyzer.html [En ligne ; consulté le 05-août-2012]. (Cité en page 173.)

[Hurd *et al.* 2010] J. Hurd, A. Tomb et D. Burke. *Visualizing Information Flow through C Programs Automated Security Analysis Project C Information Flow Tool (Cift).* In Workshop on Systems Software Verification, Vancouver, BC, Canada, 2010. USENIX Association. (Cité en page 187.)

[Joye 2008] M. Joye. *On White-Box Cryptography.* In Proceedings of the 1st International Conference Security of Information and Networks, pages 7–12. Trafford Publishing, 2008. (Cité en pages 6 et 67.)

[Keil 2012] Keil. *OPTIMIZE Compiler Directive.* Rapport technique, ARM Ltd and ARM Germany GmbH., 2012. http://www.keil.com/support/man/docs/c51/c51_optimize.htm [En ligne ; consulté le 05-août-2012]. (Cité en page 78.)

[Klockwork 2012] Klockwork. *On the fly source code analysis*, 2012. http://www.klocwork.com/products/insight/?source=feature [En ligne ; consulté le 05-août-2012]. (Cité en page 173.)

[Kocher *et al.* 2011] P. Kocher, J. Jaffe, B. Jun et P. Rohatgi. *Introduction to differential power analysis.* Journal of Cryptographic Engineering, vol. 1, pages 5–27, 2011. (Cité en page 83.)

[Kocher 1996] P. C. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. In N. Koblitz, editeur, CRYPTO, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. (Cité en pages 27 et 83.)

[Kömmerling & Kuhn 1999] O. Kömmerling et M. G. Kuhn. *Design principles for tamper-resistant smartcard processors*. In USENIX Workshop on Smartcard Technology, pages 9–20, Chicago, Illinois, USA, 1999. (Cité en page 85.)

[Kordy *et al.* 2010] B. Kordy, S. Mauw, M. Melissen et P. Schweitzer. *Attack-defense trees and two-player binary zero-sum extensive form games are equivalent*. In Proceedings of the First international conference on Decision and game theory for security - GameSec, GameSec'10, pages 245–256, Berlin, Heidelberg, 2010. Springer-Verlag. (Cité en page 108.)

[Kurtz 2004] B. Kurtz. *SoftViz : A Run-time Software Visualization Environment*. PhD thesis, Worcester Polytechnic Institute, 2004. (Cité en page 188.)

[Kushner 2012] D. Kushner. *Machine Politics : NewYorker interview of George Hotz*, 2012. http://www.newyorker.com/reporting/2012/05/07/120507fa_fact_kushner?currentPage=all [En ligne ; consulté le 05-août-2012]. (Cité en page 1.)

[Lancia 2011] J. Lancia. *Un framework de fuzzing pour cartes à puce : application aux protocoles EMV*. In Symposium sur la Sécurité des Technologies de l'Information et des Communications, Rennes, France, 2011. (Cité en pages 18 et 44.)

[Lancia 2012] J. Lancia. *Compromission d'une application bancaire JavaCard par attaque logicielle*. In Symposium sur la Sécurité des Technologies de l'Information et des Communications, Rennes, France, 2012. (Cité en page 87.)

[Lanet 2000] J.-L. Lanet. *Are Smart Cards the Ideal Domain for Applying Formal Methods ?* In J. P. Bowen, S. Dunne, A. Galloway et S. King, editeurs, First International Conference of B and Z Users - ZB, volume 1878 of *Lecture Notes in Computer Science*, pages 363–373, York, UK, 2000. Springer. (Cité en pages 52 et 176.)

[Leroy 2003] X. Leroy. *Computer Security from a Programming Language and Static Analysis Perspective*. In P. Degano, editeur, European Symposium on Programming, volume 2618 of *Lecture Notes in Computer Science*, pages 1–9, Warsaw, Poland, 2003. Springer. (Cité en page 2.)

[Leroy 2004] X. Leroy. *Exploiting type systems and static analyses for smart card security*. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet et T. Muntean, editeurs, Construction and Analysis of Safe, Secure, and Interoperable Smart Devices - CASSIS, volume 3362 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2004. (Cité en page 77.)

[Leveugle 2007] R. Leveugle. *Early Analysis of Fault-based Attack Effects in Secure Circuits*. IEEE Transactions on Computers, vol. 56, no. 10, pages 1431–1434, 2007. (Cité en page 110.)

[Litwak 1999] R. Litwak. *Fonctionnement interne d'un microprocesseur*, 1999. http://rlitwak.plil.fr/Cours_MuP/rlit321.html [En ligne ; consulté le 05-août-2012]. (Cité en pages 24 et 211.)

[Ly *et al.* 2008] O. Ly, P. Andouard et D. Rouillard. *VisAA : Visual Analyzer for Assembler*. In Logic and Automata, pages 221–225, Tozeur, Tunisia, 2008. (Cité en page 188.)

[Machemie *et al.* 2011] J.-B. Machemie, C. Mazin, J.-L. Lanet et J. Iguchi-Cartigny. *SmartCM A Smart Card Fault Injection Simulator*. In IEEE International Workshop on Information Forensics and Security, pages 1–6, Foz do Iguaçu, Portugal, 2011. (Cité en page 145.)

[Madeira *et al.* 2000] H. Madeira, D. Costa et M. Vieira. *On the Emulation of Software Faults by Software Fault Injection*. In International Conference on Dependable Systems and Networks, pages 417–426, New York city, NY, USA, 2000. (Cité en pages 144, 145 et 146.)

[Malacaria & Hankin 1999] P. Malacaria et C. Hankin. *Non-Deterministic Games and Program Analysis : An Application to Security*. In IEEE Symposium on Logic in Computer Science - LICS, pages 443–452, Trento, Italy, 1999. IEEE Computer Society Press. (Cité en page 108.)

[Manshaei *et al.* 2012] M. Manshaei, Q. Zhu, T. Alpcan, T. Basar et J.-P. Hubaux. *Game Theory Meets Network Security and Privacy*. Rapport technique, Ecole Polytechnique Fédérale de Lausanne, Vol. 45, Issue # 3, September 2013, 2012. (Cité en page 108.)

[Marche & Rousset 2006] C. Marche et N. Rousset. Verification of JAVA CARD Applets Behavior with Respect to Transactions and Card Tears. IEEE, 2006. (Cité en page 105.)

[Marjamäki 2012] D. Marjamäki. *A tool for static C/C++ code analysis*, 2012. http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page [En ligne ; consulté le 05-août-2012]. (Cité en page 174.)

[McCarty 2004] B. McCarty. Selinux : Nsa's open source security enhanced linux. O'Reilly Media, Inc., 2004. (Cité en page 50.)

[McCormac *et al.* 1995] J. McCormac, K. Vikor, M. Williams, R. Vreeman, L. Sugoy et B. McIlwrath. *Decoding Pay TV (European Scrambling Systems)*, 1995. http://cd.textfiles.com/hackersencyc/ETC/HARDWARE/FAQ2.HTM [En ligne ; consulté le 05-août-2012]. (Cité en page 86.)

[Microsoft 2002] Microsoft. *The STRIDE Threat Model*, 2002. http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx [En ligne ; consulté le 05-août-2012]. (Cité en page 108.)

[Miller *et al.* 2001] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii et F. I. Popovici. *Playing Inside the Black Box : Using Dynamic Instrumentation to Create Security Holes*. Parallel Processing Letters, vol. 11, no. 2/3, pages 267–280, 2001. (Cité en page 93.)

[Moebius *et al.* 2009] N. Moebius, K. Stenzel et W. Reif. *Generating formal specifications for security-critical applications - A model-driven approach*. In Workshop on Software Engineering for Secure Systems - ICSE, IWSESS '09, pages 68–74, Washington, DC, USA, 2009. IEEE Computer Society. (Cité en page 79.)

[Monate & Signoles 2008] B. Monate et J. Signoles. *Slicing for Security of Code*. In P. Lipp, A.-R. Sadeghi et K.-M. Koch, editeurs, Trusted Computing - Challenges and Applications - TRUST, volume 4968 of *Lecture Notes in Computer Science*, pages 133–142, Olso, Norway, 2008. Springer. (Cité en page 139.)

[Muchnick 1998] S. S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann, 1998. (Cité en page 120.)

[Murdoch *et al.* 2010] S. J. Murdoch, S. Drimer, R. J. Anderson et M. Bond. *Chip and PIN is Broken*. In IEEE Symposium on Security and Privacy, pages 433–446, Berleley/Oakland, CA, USA, 2010. IEEE Computer Society. (Cité en pages 16 et 28.)

[Myers *et al.* 2004] A. C. Myers, A. Sabelfeld et S. Zdancewic. *Enforcing Robust Declassification.* In Workshop on Computer Security Foundations - CSFW, pages 172–186, Pacific Grove, CA, USA, 2004. IEEE Computer Society. (Cité en pages 76 et 77.)

[Mysore *et al.* 2008] S. Mysore, B. Mazloom, B. Agrawal et T. Sherwood. *Understanding and visualizing full systems with data flow tomography.* In S. J. Eggers et J. R. Larus, editeurs, Architectural Support for Programming Languages and Operating Systems - ASPLOS, pages 211–221, Seattle, WA, USA, 2008. ACM. (Cité en page 188.)

[Nethercote & Seward 2007] N. Nethercote et J. Seward. *Valgrind : a framework for heavyweight dynamic binary instrumentation.* In Programming Language Design and Implementation - PLDI, PLDI'07, pages 89–100, San Diego, CA, USA, 2007. ACM. (Cité en page 188.)

[Nguyen 2011] M. H. Nguyen. *Sécurisation de processeurs vis-à-vis des attaques par faute et par analyse de la consommation.* PhD thesis, Université Pierre et Marie Curie (UPMC), 2011. Type : Thèse de Doctorat – Soutenue le : 2011-09-21 – Dirigée par : Drach-temam, Nathalie – Encadrée par : ROBISSON Bruno. (Cité en page 85.)

[Nori *et al.* 2009] A. V. Nori, S. K. Rajamani, S. Tetali et A. V. Thakur. *The YogiProject : Software Property Checking via Static Analysis and Testing.* In S. Kowalewski et A. Philippou, editeurs, Tools and Algorithms for the Construction and Analysis of Systems - TACAS, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181, York, UK, 2009. Springer. (Cité en pages 139 et 167.)

[Otto 2005] M. Otto. *Fault Attacks and Countermeasures.* PhD thesis, University of Paderborn, 2005. (Cité en page 86.)

[Parasoft 2012] Parasoft. *Parasoft Application Security Solution*, 2012. http://www.parasoft.com [En ligne ; consulté le 05-août-2012]. (Cité en page 173.)

[Pietriga 2005] E. Pietriga. *A Toolkit for Addressing HCI Issues in Visual Language Environments.* In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 145–152, Dallas, TX, USA, 2005. IEEE Computer Society. (Cité en page 182.)

[Pistoia *et al.* 2007] M. Pistoia, A. Banerjee et D. a. Naumann. *Beyond Stack Inspection : A Unified Access-Control and Information-Flow Security Model.* IEEE Symposium on Security and Privacy - SP, pages 149–163, 2007. (Cité en page 107.)

[Prouff & Rivain 2009] E. Prouff et M. Rivain. *Combining Information Theory and Side Channels to Break Secure Implementations.* In e-Smart, Nice, France, 2009. (Cité en page 78.)

[Quisquater & Kouene 2002] J.-J. Quisquater et F. Kouene. *Side Channel Attacks*, 2002. (Cité en pages 84 et 85.)

[Quisquater & Samyde 2001] J.-J. Quisquater et D. Samyde. *ElectroMagnetic Analysis (EMA) : Measures and Counter-measures for Smart Cards.* In I. Attali et T. Jensen, editeurs, Smart Card Programming and Security, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag Berlin Heidelberg, Cannes, France, 2001. (Cité en page 83.)

[Quisquater & Samyde 2002] J.-J. Quisquater et D. Samyde. *Eddy current for Magnetic Analysis with Active Sensor.* In e-Smart, Nice, France, 2002. (Cité en page 85.)

[Quisquater *et al.* 1989] J.-J. Quisquater, M. Quisquater, M. Quisquater, M. Quisquater, L. C. Guillou, M. A. Guillou, G. Guillou, A. Guillou, G. Guillou, S. Guillou et T. A. Berson. *How to Explain Zero-Knowledge Protocols to Your Children.* In G. Brassard, editeur, CRYPTO, volume 435 of *Lecture Notes in Computer Science*, pages 628–631, London, UK, 1989. Springer. (Cité en page 4.)

[Rankl & Effing 2003] W. Rankl et W. Effing. Smart card handbook, pages 667–734. John Wiley & Sons, 2003. Overview about Attacks on Smart Cards. (Cité en pages 2 et 85.)

[Riscure 2012] Riscure. *Riscure Security Tools*, 2012. http://www.riscure.com/tools/inspector [En ligne ; consulté le 05-août-2012]. (Cité en page 84.)

[Rook 2011] D. Rook. *Agnitio tool for security code review*, 2011. http://sourceforge.net/projects/agnitiotool/ [En ligne ; consulté le 05-août-2012]. (Cité en page 175.)

[Rothbart *et al.* 2004] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger et A. Muehlberger. *High Level Fault Injection for Attack Simulation in Smart Cards.* In 13th Asian Test Symposium, ATS '04, pages 118–121, Kenting, Taiwan, 2004. IEEE Computer Society. (Cité en page 145.)

[Sabelfeld & Myers 2003] A. Sabelfeld et A. C. Myers. *Language-based information-flow security.* IEEE Journal on Selected Areas in Communications, vol. 21, no. 1, pages 5–19, 2003. (Cité en page 51.)

[Samyde *et al.* 2002] D. Samyde, S. Skorobogatov, R. Anderson et J.-J. Quisquater. *On a new way to read data from memory.* In First International IEEE Security in Storage Workshop, pages 65–69, Greenbelt, MD, USA, 2002. IEEE Comput. Soc. (Cité en page 84.)

[Schellhorn *et al.* 2002] G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel et D. Toll. *Verified formal security models for multiapplicative smart cards.* Journal of Computer Security, vol. 10, no. 4, 2002. (Cité en page 50.)

[Schwan 2008] M. Schwan. *Specification and verification of security policies for smart cards.* PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät II, 2008. (Cité en page 50.)

[SCL 2012] *Side Channel Lounge*, 2012. http://imperia.rz.rub.de:9085/en_sclounge.html [En ligne ; consulté le 05-août-2012]. (Cité en page 84.)

[Sere *et al.* 2011] A. Sere, J.-L. Lanet et J. Iguchi-Cartigny. *Evaluation of Countermeasures Against Fault Attacks on Smart Cards.* International Journal of Security and Its Applications, vol. 5, no. 2, 2011. (Cité en page 145.)

[Shabtai *et al.* 2010] A. Shabtai, Y. Fledel et Y. Elovici. *Securing Android-Powered Mobile Devices Using SELinux.* IEEE Security & Privacy, vol. 8, no. 3, pages 36–44, 2010. (Cité en page 50.)

[Shacham 2007] H. Shacham. *The geometry of innocent flesh on the bone : return-into-libc without function calls (on the x86).* In Conference on Computer and communications Security, CCS '07, pages 552–561, Alexandria, VA, USA, 2007. ACM. (Cité en page 100.)

[Skorobogatov & Anderson 2002] S. P. Skorobogatov et R. J. Anderson. *Optical Fault Induction Attacks.* In B. S. Kaliski Jr., Ç. K. Koç et C. Paar, editeurs, Cryptographic Hardware and Embedded Systems - CHES, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12, Redwood Shores, CA, USA, 2002. Springer. (Cité en pages 28 et 85.)

[Skorobogatov 2005] S. P. Skorobogatov. *Semi-invasive attacks – A new approach to hardware security analysis*. Rapport technique UCAM-CL-TR-630, University of Cambridge, Computer Laboratory, 2005. (Cité en page 86.)

[Skorobogatov 2010] S. Skorobogatov. *Optical Fault Masking Attacks*. In Workshop on Fault Diagnosis and Tolerance in Cryptography, Fault Diagnosis and Tolerance in Cryptography, pages 23–29, Santa Barbara, California, USA, 2010. IEEE Computer Society. (Cité en page 86.)

[Spinroot 2012] Spinroot. *Static Source Code Analysis Tools for C*, 2012. http://spinroot.com/static/ [En ligne ; consulté le 05-août-2012]. (Cité en page 174.)

[Srivatanakul *et al.* 2005] T. Srivatanakul, J. A. Clark et F. Polack. *Stressing Security Requirements : Exploiting the Flaw Hypothesis Method with Deviational Techniques*. In Symposium on Requirements Engineering for Information Security, Paris, France, 2005. (Cité en page 143.)

[Tarnovsky 2010] C. Tarnovsky. *Deconstructing a Secure Processor*. In Black Hat, Washington, DC, USA, 2010. https://media.blackhat.com/bh-dc-10/video/Tarnovsky_Chris/BlackHat-DC-2010-Tarnovsky-DeconstructProcessor-video.m4v [En ligne ; consulté le 05-août-2012]. (Cité en page 28.)

[Teuwen 2010] P. Teuwen. *How to Make Smartcards Resistant to Hackers' Lightsabers ?* In J. Guajardo and B. Preneel and A.-R. Sadeghi and P. Tuyls, editeur, Foundations for Forgery-Resilient Cryptographic Hardware, pages 1–8, Dagstuhl, Germany, 2010. (Cité en page 90.)

[Thorn 1999] T. Thorn. *Vérfication de politiques de sécurité par analyse de programmes*. PhD thesis, Université de Rennes I, Ifsic, Irisa, 1999. (Cité en page 51.)

[van Heesch 2008] D. van Heesch. *Doxygen : Source code documentation generator tool*, 2008. http://www.stack.nl/~dimitri/doxygen/ [En ligne ; consulté le 05-août-2012]. (Cité en page 176.)

[van Woudenberg *et al.* 2011] J. G. van Woudenberg, M. F. Witteman et F. Menarini. *Practical Optical Fault Injection on Secure Microcontrollers*. In Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 91–99, Los Alamitos, CA, USA, 2011. IEEE Computer Society. (Cité en page 27.)

[Verbauwhede *et al.* 2011] I. Verbauwhede, D. Karaklajic et J.-M. Schmidt. *The Fault Attack Jungle - A Classification Model to Guide You*. In Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 3–8, Nara, Japan, 2011. Ieee. (Cité en page 85.)

[von Oheimb *et al.* 2003] D. von Oheimb, G. Walter et V. Lotz. *A Formal Security Model of the Infineon SLE 88 Smart Card Memory Managment*. In E. Snekkenes et D. Gollmann, editeurs, 8th European Symposium on Research in Computer Security - ESORICS, volume 2808 of *Lecture Notes in Computer Science*, pages 217–234, Gjøvik, Norway, 2003. Springer. (Cité en page 52.)

[Wagner 2004] D. Wagner. *Cryptanalysis of a Provably Secure CRT-RSA Algorithm*. In 11th ACM conference on Computer and communications security, pages 92–97, Washington, DC, USA, 2004. (Cité en page 81.)

[Wikipedia 2012a] Wikipedia. *EMV Vulnerabilities*, 2012. http://en.wikipedia.org/wiki/EMV#Vulnerabilities [En ligne ; consulté le 05-août-2012]. (Cité en page 18.)

[Wikipedia 2012b] Wikipedia. *George Hotz*, 2012. http://en.wikipedia.org/wiki/George_Hotz [En ligne ; consulté le 05-août-2012]. (Cité en page 1.)

[Wikipedia 2012c] Wikipedia. *Norme Iso 14443*, 2012. http://en.wikipedia.org/wiki/ISO/IEC_14443 [En ligne ; consulté le 05-août-2012]. (Cité en page 16.)

[Wikipedia 2012d] Wikipedia. *Norme Iso 7816*, 2012. http://en.wikipedia.org/wiki/ISO/IEC_7816 [En ligne ; consulté le 05-août-2012]. (Cité en page 16.)

[Williams *et al.* 2005] N. Williams, B. Marre, P. Mouy et M. Roger. *PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis.* In M. D. Cin, M. Kaâniche et A. Pataricza, editeurs, Fifth European Dependable Computing Conference, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292, Budapest, Hungary, 2005. Springer. (Cité en page 181.)

[Wolf 1994] W. Wolf. *Hardware-Software Co-Design of Embedded Systems.* Proceedings of the IEEE, vol. 82, no. 7, pages 967–989, 1994. (Cité en page 190.)

[Zanotti 2002] M. Zanotti. *Security Typings by Abstract Interpretation.* In 9th International Symposium on Static Analysis - SAS, SAS '02, pages 360–375, Madrid, Spain, 2002. Springer-Verlag. (Cité en page 52.)

Glossaire

8051 Intel 8051 ou 8051 est un microcontrôleur (MC) développé par Intel en 1980 pour être utilisé dans des produits embarqués. C'est encore une architecture populaire ; de nombreux microcontrôleurs plus récents incorporent un coeur 8051, complété par un certain nombre de circuits périphériques intégrés sur la même puce, et dotés de mémoires de plus grande capacité.. 9

Android Android est un système d'exploitation open source3 utilisant le noyau Linux, pour smartphones, tablettes tactiles, PDA et terminaux mobiles conçu par Android, une startup rachetée par Google, et annoncé officiellement le 5 novembre 20074. D'autres types d'appareils possédant ce système d'exploitation existent, par exemple des téléviseurs et des tablettes.. 50

ANSSI L'Agence nationale de la sécurité des systèmes d'information est une agence chargée de la sécurité informatique rattachée au Secrétariat général de la défense et de la sécurité nationale.. 10

APDU APDU (Application Protocol Data Unit) est une trame unitaire normalisée de communication entre le terminal et la carte.. 16

Canal+ Canal+ (stylisée CANAL+) est une chaine de télévision généraliste française privée à péage axée sur le cinéma et le sport. Toute première chaine privée à péage en France, elle appartient au groupe Canal+ (filiale du groupe Vivendi).. 9

CESTI Centre d'évaluation de la sécurité des technologies de l'information est un prestataire de service, indépendant, chargé d'évaluer la conformité d'un produit aux Critères communs.. 10

Checksum Un checksum ou somme de contrôle est une forme de contrôle par redondance. Cette empreinte est une valeur utilisée pour s'assurer qu'une donnée est mémorisée ou transmise sans erreur.. 25

DLL En informatique, une bibliothèque ou bibliothèque de programmes est un ensemble de fonctions utilitaires, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire. Les fonctions sont regroupées de par leur appartenance à un même domaine conceptuel (mathématique, graphique, tris, etc). Les bibliothèques logicielles se distinguent des exécutables dans la mesure où elles ne représentent pas une application. Elles ne sont pas complètes, elles ne possèdent pas l'essentiel d'un programme comme une fonction principale et par conséquent ne peuvent pas être exécutées directement. 170

ECC En cryptographie, les courbes elliptiques, des objets mathématiques, peuvent être utilisées pour des opérations asymétriques comme des échanges de clés sur un canal non-sécurisé ou un chiffrement asymétrique, on parle alors de cryptographie sur les courbes elliptiques ou ECC (de l'acronyme anglais Elliptic curve cryptography). L'usage des courbes elliptiques en cryptographie a été suggéré, de manière indépendante, par Neal Koblitz et Victor Miller en 1985.. 4

EMVCo Europay Mastercard Visa, abrégé par le sigle EMV, est depuis 1995 le standard international de sécurité des cartes de paiement (cartes à puce). Il tire son nom des organismes fondateurs : Europay International (absorbé par Mastercard en 2002), MasterCard International et Visa International. 16

FIB La Sonde ionique focalisée, plus connue sous le nom du sigle anglais FIB (Focused ion beam), est un instrument scientifique qui ressemble au microscope électronique à balayage (MEB). Mais là où le MEB utilise un faisceau d'électrons focalisés pour faire l'image d'un échantillon, la "FIB" utilise un faisceau d'ions focalisés, généralement du gallium. Il est en effet facile de construire une source à métal liquide (LMIS, de l'anglais liquid metal ion source. Contrairement aux MEB, les FIB sont destructives. Par conséquent, leur domaine d'applications est plus la microfabrication que la microscopie. Les principaux domaines d'applications sont la science des matériaux et en particulier le domaine des semiconducteurs et des circuits intégrés.. 28

fuzzing Le fuzzing est une technique pour tester des logiciels. L'idée est d'injecter des données aléatoires dans les entrées d'un programme. Si le programme échoue (par exemple en crashant ou en générant une erreur), alors il y a des défauts à corriger.. 18, 44

GIE CB Le Groupement des Cartes Bancaires CB est un groupement d'intérêt économique privé qui réunit la plupart des établissements financiers français dans le but d'assurer l'interbancarité des cartes de paiement.. 6

Javacard Java Card fait référence à la technologie développée par Sun maintenant Oracle qui permet l'exécution d'application sous forme d'applet Java sur un support telle qu'une carte à puce Java Card est un environnement d'exécution Java destiné aux applications pour Carte à puce. Cette technologie fournit un environnement sécurisé pour les applications qui fonctionnent sur ce support de capacité mémoire et de traitement limités. De multiples applications peuvent être déployées avant et même après que la Carte à puce a été fournie à l'utilisateur final. Les applications écrites dans le langage de programmation Java peuvent être exécutées en toute sécurité sur l'ensemble des types de cartes disponibles sur le marché.. 9, 23

Linux Linux ou GNU/Linux, est un système d'exploitation libre fonctionnant avec le noyau Linux. C'est une implémentation libre du système UNIX respectant les spécifications POSIX. Linux est le système le plus utilisé sur les super-ordinateurs et les smartphones. Sur les serveurs informatiques, le marché est partagé avec les autres Unix et Windows. Il est largement utilisé comme système embarqué dans les appareils électroniques : télévision, modem, GPS, etc.. 50

Mastercard MasterCard Worldwide (NYSE : MA) est une entreprise américaine de système de paiement dont le siège est à Purchase, New York. Jusqu'en 2006, c'était une société coopérative détenue par plus de 25 000 institutions financières. Depuis, elle est une société cotée en Bourse. MasterCard est aussi la marque de cartes de crédit, un des produits de cette société.. 9

Mifare MIFARE est une des technologies de carte à puce sans contact les plus répandues dans le monde avec 3,5 milliards de cartes et 40 millions de modules de lecture/encodage. La marque, lancée par Philips, est propriété de la société NXP.. 9

Moneo Moneo (parfois écrit Monéo) est le seul système de porte-monnaie électronique utilisé en France en 2011. Il peut être matérialisé sur une carte bancaire ou sur une carte dédiée à cet usage. Les montants concernés par les transactions sont de l'ordre de la petite monnaie (distributeur, automate, café, musique, boulangerie, journal, parcmètre, etc.).. 9

MultOS MULTOS ("Multiple Operating System") est un système d'exploitation qui permet à l'aide d'une machine virtuelle d'éxécuter plusieurs applications différentes de manière sécurisée sur la même carte à puce.. 10

Passe Navigo Un passe Navigo est une carte à puce sans contact, utilisant la technologie RFID (Radio Frequency IDentification, radio-identification) ou plus précisément NFC (Near Field Communication, communication en champ proche), qui sert de support pour certains forfaits d'abonnement utilisables dans les transports en Île-de-France sur les réseaux RATP, SNCF et Optile. Sa mise en oeuvre est supervisée par le STIF, qui est propriétaire de la marque.. 9

OpenCard La plate-forme OpenCard est un middleware implémenté en Java.. 10

PIN Personal Identification Number se composant souvent de 4 chiffres, sa connaissance permet avec la possession d'une carte à puce associée d'authentifier le porteur auprès d'une banque et de réaliser des transactions bancaires. 8

SIM La carte SIM (de l'anglais Subscriber Identity Module) est une puce contenant un microcontrôleur et de la mémoire. Elle est utilisée en téléphonie mobile pour stocker les informations spécifiques à l'abonné d'un réseau mobile, en particulier pour les réseaux de type GSM ou UMTS. Elle permet également de stocker des applications de l'utilisateur, de son opérateur ou dans certains cas de tierces parties.. 9

VISA VISA est une marque de carte de paiement de la Visa International Service Association. VISA est aussi le nom de l'entreprise commune composée de quelque 21 000 sociétés financières (banques, sociétés de crédit).. 9

Table des Figures

1.1 Schéma d'une authentification hors ligne ne nécessitant pas de participation de la banque dans la transaction [Bayart 2012] . . . 7
1.2 Schéma d'une authentification en ligne authentifiant la carte auprès de la banque au moment de la transaction [Bayart 2012] . . . 7
1.3 Schéma d'authentification bancaire . . . 8
1.4 Schéma d'authentification téléphonique . . . 9
1.5 Description simplifiée des différentes étapes dans le processus de développement du logiciel d'une carte à puce . . . 13
1.6 Une simulation logicielle des tests de sécurité matérielle entraine une économie de tests physiques plus longs et plus couteux . . . 14
1.7 Itération préliminaire entre la phase de développement et la phase de test logiciel simplifiant la phase de test matériel . . . 14
1.8 Schéma simplifié d'une carte . . . 19
1.9 Schéma d'une carte [Consultants 2009] . . . 20
1.10 Illustration du micromodule . . . 20
1.11 Architecture interne du microprocesseur [Litwak 1999] . . . 24
1.12 Vue d'une puce et des différents composants mémoire [Guilley 2007] . . . 26
1.13 Banc d'attaque laser, source : Oberthur Technologies . . . 29
1.14 Axes d'approche du problème . . . 44
1.15 La place de la sécurité dans le contexte embarqué . . . 45

2.1 Évolution des informations à l'intérieur des conteneurs correspondant au code du listing 2.5 . . . 57
2.2 Avec une méthode d'observation et un point d'observation certains conteneurs, ainsi que l'information qu'ils abritent, sont visibles . . . 58
2.3 La confidentialité est directement compromise si une information se trouve dans un conteneur visible . . . 59
2.4 Évolution des informations à l'intérieur des conteneurs correspondant au code du listing 2.6 . . . 60
2.5 Exemple de durée de vie d'un conteneur et durée de vie d'une information à l'intérieur d'un conteneur . . . 63

3.1 Fuite d'informations par les canaux d'observation auxiliaires . . . 83
3.2 Exemple d'un code C et son équivalent en assembleur 8051C . . . 92
3.3 Illustration de la notion de vol et des étapes . . . 99
3.4 Cas 2 . . . 102
3.5 Champ d'application du lemme et du théorème au niveau des blocs basiques . . . 104

5.1 Schéma de la méthodologie adoptée pour l'analyse sécuritaire . . . 142
5.2 Vue d'ensemble de la plate-forme expérimentale . . . 143
5.3 Représentation 2D des résultats donné par un oracle . . . 147

5.4 Nombre d'attaques pour chaque catégorie en fonction du délai d'expiration (en utilisant la méthode GDB) . . . 149
5.5 Résultats d'attaques permanentes réalisées en C sur le code de BZIP2 . . . 152
5.6 Résultats d'attaques permanentes réalisées en C sur le code de GZIP . . . 154
5.7 "Génome" de BZIP2 . . . 156
5.8 "Génome" de GZIP . . . 156
5.9 Comparaison des "génomes" de BZIP2 et GZIP . . . 157
5.10 Les attaques transientes injectées avec GDB dans la totalité de BZIP2 . . . 158
5.11 Classement des attaques transientes pour la fonction mainQSort3 de BZIP2 en fonction du moment où l'attaque est déclenchée . . . 158
5.12 Distribution des attaques permanentes injectée au niveau ASM / C pour les fonctions de BZIP2 de plus de 7 instructions . . . 159
5.13 Nombre d'attaques en fonction de la taille du saut exprimé en lignes de code C . 161
5.14 Classification spatiale des attaques good/bad/kill en fonction des lignes source et destination, simulées en C pour la fonction BZ2_compressBlock . . . 161
5.15 Classification spatiale des attaques good/bad/kill en fonction des lignes source et destination, simulées en C pour la fonction BZ2_blockSort . . . 162
5.16 Classification spatiale des attaques good/bad/kill en fonction des lignes source et destination, simulées en C pour la version sécurisée de BZ2_blockSort . . . 163
5.17 Classification spatiale des attaques kill/bad/error en fonction des lignes source et destination, simulées en C pour une fonction sensible du code source de d'une carte à puce . . . 165
5.18 Nombre de résultats BAD en fonction de la distance de l'attaque par saut . . . 166

6.1 Liste de vérification dans le code . . . 175
6.2 Évolution du code au cours des revues . . . 175
6.3 Exemple de documentation générée avec Doxygen . . . 176
6.4 Exemple de résultats obtenus par l'outil . . . 178
6.5 Frama-C - Analyse de code . . . 180
6.6 Frama-C - Impact d'une instruction . . . 180
6.7 Réduction progressive de la complexité du programme par interrogation du graphe d'appels . . . 183
6.8 Réduction à l'échelle d'un module puis d'un ensemble de fonctions . . . 184

Liste des Tableaux

3.1 Croisement entre la cible et les effets d'une attaque 89

4.1 Règles de typage pour les appels de fonctions 121
4.2 Exemple de valeurs pour la fonction *LineTypeAfter* avec une condition 132

5.1 Statistiques des attaques simulées sur BZIP2 159
5.2 Couverture d'attaques 159

6.1 Croisement entre les fonctions et les globales utilisées 186

Liste des Listings

1.1 Exemple d'une implémentation d'authentification basique avec une tâche fonctionnelle sécuritaire sensible mais sans code contre attaques physiques 32
1.2 Exemple d'une implémentation d'authentification basique avec une tache fonctionnelle sécuritaire sensible avec quelques défenses additionnelles contre les attaques physiques 35
1.3 Condition non sécurisée 38
1.4 Condition sécurisée naivement 38
1.5 Implémentation C d'une condition 39
1.6 Traduction de cette condition en assembleur 68000 39
1.7 Condition sécurisée avec une première méthode 40
1.8 Requête schématique 42
1.9 Exemple simple d'une fonction avec doublement sécuritaire qui n'a pas d'impact sur la finalité fonctionnelle 43
2.1 Cas particulier 52
2.2 Exemple d'une implémentation basique d'un mécanisme d'authentification 53
2.3 Transfert d'information par conteneur 55
2.4 Évolution d'une information par affectation directe de valeur 55
2.5 Transfert d'information sans attaque 56
2.6 Transfert d'information avec attaque 59
2.7 Exemple de transfert d'information dans des conteneurs 63
2.8 Exemple d'une implémentation d'authentification basique avec une tache fonctionnelle sécuritaire sensible mais sans code contre attaques physiques 65
2.9 Exemple d'une implémentation d'authentification basique avec une tache fonctionnelle sécuritaire sensible mais sans code contre attaques physiques 67
2.10 déchiffrement d'une information confidentielle 68
2.11 Comparaison chiffrée d'une information confidentielle 68

2.12 Zone de garantie souhaitée pour l'intégrité du conteneur card_pin sous attaque . 70
2.13 Intégrité d'exécution d'une condition en C 74
2.14 Intégrité d'exécution d'une boucle en C 74
2.15 Intégrité d'exécution d'un appel de fonction en C 75
2.16 Intégrité d'exécution d'un appel de fonction en C 75
3.1 Exemple de code 87
3.2 Modification de la valeur d'une variable 87
3.3 Attaque par saut 88
3.4 Attaque par non exécution 88
3.5 Exemple d'une injection d'un NOP qui permet la découverte du PIN en 9999 essais 91
3.6 Exemple d'une fonction C 92
3.7 Équivalent assembleur de la fonction exemple 92
3.8 Simulation 1 94
3.9 Simulation 2 94
3.10 Simulation 3 94
3.11 Simulation 4 94
3.12 Simulation 5 95
3.13 Simulation 6 95
3.14 Simulation 7 95
3.15 Code original 96
3.16 Code avec attaques 96
3.17 Exemple d'une attaque simple à conséquences multiple 105
3.18 Code original 105
3.19 Conséquence multiples : écriture + saut 106
3.20 Conséquence multiples : écriture + écriture 106
3.21 Version sécurisée d'une condition vulnérable à une attaque par saut simple 106
3.22 Traduction en assembleur du code C du listing 3.21 107
3.23 Traduction en assembleur du code C du listing 3.21 sous attaque 107
3.24 Equivalent C du code assembleur attaqué 107
4.1 Exemple de code C pour une transaction bancaire 112
4.2 Exemple de code C pour une transaction bancaire avec des annotations ACSL . . 113
4.3 Analyse Frama-C du listing 4.2 114
4.4 Patch pour l'audit de la variable globale number1 116
4.5 Extrait du fichier d'audit 116
4.6 Injection des annotations ACSL à partir des informations collectées 117
4.7 Analyse Frama-C avec les nouvelles informations 118
4.8 Analyse Frama-C avec une attaque de valeur simulée 119
4.9 Code d'exemple illustrant les fonctions de typage 122
4.10 Exemple de code contenant un if 124
4.11 i en dehors de la condition — x non utilisé 128
4.12 i en dehors de la condition — x dans condition 129
4.13 i en dehors de la condition — x dans then 129
4.14 i en dehors de la condition — x dans then et else 129
4.15 i dans then — x non utilisé dans then 130
4.16 i dans then — x utilisé dans then avant et après i 130

4.17 i dans then — x utilisé dans then après i . . . 130
4.18 i dans then — x utilisé dans then avant i . . . 130
4.19 i en dehors de la boucle — x non utilisé . . . 132
4.20 i en dehors de la boucle — x utilisé dans la condition . . . 133
4.21 i en dehors de la boucle — x utilisé dans la boucle . . . 133
4.22 i dans la boucle — x utilisé dans la boucle après i . . . 133
4.23 i dans la boucle — x utilisé dans la boucle avant i . . . 134
4.24 i dans la boucle — x utilisé dans la condition . . . 134
4.25 i dans la boucle — x non utilisé . . . 134
4.26 Fichiers créés par le prototyle . . . 137
5.1 Exemple d'une zone de code sautée . . . 148
5.2 Implémentation d'une contre-mesure pour la fonction BZ2_blockSort ne nécessitant pas de décalage de la numérotation des lignes du code de départ . . . 163
6.1 Exemple de code pour catégorisation par mots clés . . . 177
6.2 Exemple de code pour catégorisation par mots clés . . . 179
6.3 Explorateur interactif de code source en ligne de commande . . . 182
6.4 Exemple de chemins obtenus dans le graphe d'appel du programme . . . 182
6.5 Commandes utilisées pour créer les graphes . . . 185
6.6 Exemple de recherche de formes de code interprocédurale . . . 185
6.7 Exemple de code source pour recherche de formes de code interprocédurale . . . 186

Liste des Algorithmes

1 Calcul de TypeAfter pour un code séquentiel . . . 121
2 Calcul de *LineTypeAfter* pour un code linéaire . . . 122
3 Fonction TypeAfter pour les conditions . . . 126
4 Fonction LineTypeAfter pour les conditions . . . 127
5 Code à ajouter à la définition de TypeAfter . . . 135
6 Modifications à apporter à LineTypeAfter prenant en compte les boucles . . . 136

Colophon

Cette thèse a été réalisée avec LaTeX et Vim à partir d'un template par Olivier Commowick. La fonte utilisée pour les listing est `DejaVu Sans Mono`. Le glossaire a été créé à partir de Wikipédia. Les images proviennent de `http://openclipart.org`.

Xavier KAUFFMANN-TOURKESTANSKY

Analyses sécuritaires de codes de carte à puce sous attaques physiques simulées

Cette thèse s'intéresse aux effets des attaques par fautes physiques sur le code d'un système embarqué en particulier la carte à puce. De telles attaques peuvent compromettre la sécurité du système en donnant accès à des informations confidentielles, en compromettant l'intégrité de données sensibles ou en perturbant le fonctionnement pendant l'exécution. Dans cette thèse, nous décrivons des propriétés de sécurité permettant d'exprimer les garanties du système et établissons un modèle d'attaque de haut niveau définissant les capacités d'un attaquant à modifier le système. Ces propriétés et ce modèle nous servent à vérifier la sécurité du code par analyse statique ou test dynamique, combinés avec l'injection d'attaques, simulant les conséquences logicielles des fautes physiques. Deux méthodologies sont ainsi développées afin de vérifier le comportement fonctionnel du code sous attaques, tester le fonctionnement des sécurités implémentées et identifier de nouvelles attaques. Ces méthodologies ont été mises en œuvre dans un cadre industriel afin de faciliter le travail du développeur chargé de sécuriser un code de carte à puce.

Mots clés : code, sécurité, attaques physiques, carte à puce, information, propriétés, confidentialité, intégrité, analyse statique, test, injection de faute, vérification, preuve, modélisation, simulation logicielle, optimisation, réduction, outils, certification, embarqué, analyse, comportement

Security Analysis of Smart Card C Code using Simulated Physical Attacks

This thesis focuses on the effects of attacks by physical faults on embedded source code specifically for smart cards. Such attacks can compromise the security of the system by providing access to confidential information, compromising the integrity of sensitive data or disrupting the execution flow. In this thesis, we describe security properties to express security guarantees on the system. We also offer an attack model defining at high level an attacker's ability to disrupt the system. With these properties and model, we check the source code security against physical attacks. We use static analysis and dynamic testing, combined with attack injection to simulate the consequences of physical faults at software level. Two techniques are created to stress the functional behavior of the code under attack, test the reliability of built-in security countermeasures and identify new threats. These techniques were implemented in a framework to help developers secure their source code in an industrial environment.

Keywords : code, security, physical attacks, smart card, information, security properties, confidentiality, integrity, static analysis, testing, fault injection, verification, proof, hardware modeling, software simulation, optimization, reduction, tools, security model, certification, embedded, analysis, behavior

OBERTHUR TECHNOLOGIES

Laboratoire d'Informatique Fondamentale d'Orléans
Rue Léonard de Vinci
B.P. 6759 F-45067 ORLEANS Cedex 2

OBERTHUR TECHNOLOGIES