N° Ordre ……. / FS / UMBB / 2019

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ M'HAMED BOUGARA DE BOUMERDES

Faculté des **S**ciences

Thèse de Doctorat

Présentée par

HAMADOUCHE Samiya

Filière : Informatique
Option : Ingénierie des Logiciels et Traitement de l'Information

---

Construction de code malveillant activable par attaque en faute pour support d'exécution sécurisé et contre-mesure associée

---

Devant le jury :

M. BADACHE Nadjib	Professeur USTHB	Président
M. HAMADOUCHE M'hamed	Professeur UMBB	Examinateur
M. RIAHLA Mohamed Amine	MCA UMBB	Examinateur
M. MEZGHICHE Mohamed	Professeur UMBB	Directeur
M. LANET Jean-Louis	Professeur INRIA (France)	Co-directeur

Année Universitaire : 2019/2020

Remerciements

Arrivée au terme de cette thèse, il m'est particulièrement agréable d'exprimer ma gratitude et mes remerciements à tous ceux qui, par leurs connaissances, leur soutien et leurs conseils, m'ont aidé à sa réalisation.

Je tiens tout d'abord à remercier mon directeur de thèse Monsieur MEZGHICHE Mohamed, Professeur à l'Université de Boumerdès, pour m'avoir accueilli au sein du laboratoire LIMOSE où il m'a assuré un cadre favorable de travail. Je le remercie pour son écoute attentive et ininterrompue, ses conseils avisés et ses qualités humaines qui m'ont été d'une aide précieuse pour avancer. Je le remercie d'avoir toujours cru en moi. Il n'a jamais douté que j'allais parvenir à mener cette thèse à son terme. J'espère être à la hauteur de ses attentes.

Mes plus grands remerciements s'adressent aussi à mon co-directeur de thèse Monsieur LANET Jean-Louis, Professeur à l'INRIA de Rennes (France), pour la confiance qu'il m'a accordé en me proposant ce passionnent sujet de recherche. Ses conseils et ses remarques toujours très pertinents m'ont été d'une grande aide pour avancer. Je le remercie de m'avoir accueilli chaleureusement dans son équipe SSD à l'Université de Limoges puis au laboratoire LHS de l'INRIA de Rennes durant mes séjours scientifiques. L'expérience que j'ai pu acquérir en travaillant avec lui ainsi que les membres de son équipe, depuis mes travaux de Magister, sera sans doute bénéfique pour moi sur le plan scientifique, professionnel et relationnel.

J'adresse aussi mes vifs remerciements à Monsieur BADACHE Nadjib, Professeur à USTHB, de m'avoir fait l'honneur de présider mon jury de soutenance, à Monsieur HAMADOUCHE M'Hamed, Professeur à l'UMBB, et Monsieur RIAHLA Mohamed Amine, Maître de conférences à l'UMBB, d'avoir bien voulu examiner mon travail et faire partie de mon jury de soutenance.

Je remercie Monsieur CHAABANI Mohamed, chef du département d'informatique à l'UMBB, pour ses efforts et son aide. Ainsi que mes collègues enseignants, doctorants et personnel administratif du département d'informatique pour leur gentillesse.

J'ai pu travailler dans un cadre particulièrement agréable, grâce aux membres du laboratoire LIMOSE et particulièrement ceux de l'équipe 4 (Spécification de Logiciels et Traitement de l'Information). Je les remercie tous pour les discussions qu'on a pu avoir, leur bonne humeur, pour tous ces moments de rires et de détente inoubliables qu'on a pu partager. Et un merci spécial pour mes collègues et amis : Razika, Selma, Dhia Eddine, Hocine et Mohamed Tahar pour leur soutien amical.

Je tiens à remercier ma tante Safia pour les efforts qu'elle a fait pour relire le présent document en un temps record. Je remercie également mon amie et collègue de bureau Razika pour sa relecture minutieuse de certaines parties de cette thèse.

Mes remerciements pleins de reconnaissance et de gratitude vont aussi à toute ma famille. En particulier : ma mère, mon mari et mes sœurs, sans qui je n'en serai pas là aujourd'hui. Grâce à leur présence même dans les moments les plus difficiles et leur soutien moral sans failles, ils m'ont permis de surpasser les moments de doute et d'avancer droit vers mon objectif final. Je les remercie d'avoir cru en moi quand je n'y croyais plus, j'espère les rendre fières.

Je dédie ce mémoire

À la mémoire de mon père
qui restera à jamais dans mes pensées,
et au fond de mon cœur,
lui qui m'a toujours soutenu
et tant attendu ce jour ...

À celle qui m'est la plus chère au monde,
qui a été à mes côtés dans tous les moments difficiles
par ses sacrifices, son amour et son encouragement,
j'espère avoir réalisé aujourd'hui l'un de tes rêves,
j'espère te rendre toujours fière de moi Maman

À mon très cher mari Abdel Hamid
pour le soutien continu dont il a toujours fait preuve,
pour son amour, son encouragement et sa patience
qui ont toujours été pour moi d'un grand réconfort

À mon bout de chou Anis
qui a partagé avec moi cette aventure avant même sa naissance
et continue à la vivre avec moi à chaque instant
Ton sourire illumine ma vie et la rend plus joyeuse et pleine de sens

À mes très chères sœurs,
Sabrina et sa petite famille,
Chahrazed « You can do it Samiya, just start it !!! »
et Narimene notre chouchoute

Que dieu vous protège tous pour moi ...

ملخص

اكتسبت العناصر الآمنة مكانًا كبيرًا في حياتنا اليومية و هي موجودة في عدة أشكال.البطاقة الذكية هي العنصر الأكثر تمثيلا في عائلة العناصر الآمنة. وهي تعتبر أجهزة جد آمنة لتنفيذ التطبيقات وتخزين المعلومات. نظرًا لطبيعة المعلومات التي بحوزتها ، أصبحت البطاقات الذكية هدفًا للمهاجمين الذين يريدون الحصول على معلومات حساسة مناسبة يتم تخزينها أو حتى السيطرة على النظام. يمكن التحايل على أمان البطاقة الذكية بواسطة المعدات أو البرامج أو الهجمات المدمجة.عملنا ينتمي الى هذه الفئة. هدفنا هو تطوير ناقل هجوم جديد. في الواقع ، من خلال إتقان التفاصيل للتحايل على أمان البطاقة ، يمكننا أن نجد فيما بعد الإجراءات المضادة للحذر منها: "أفضل دفاع هو الهجوم"." جافا كارد" هي الأكثر استخدامًا ، حيث يتم اختيارها كمنصة مستهدفة لعملنا.

هدفنا هو العثور على منهجية لبناء برامج ضارة يتم تنشيطها عن طريق الهجوم بحقن الاخطاء. تتمثل الفكرة في إخفاء هذا الرمز الضار في رمز آخر (عن طريق الإنشاء) بحيث يمكن تحميله في البطاقة دون أن يتم اكتشافه بواسطة آليات الأمان المضمنة أو تحليل الرمز. بعد التحميل على البطاقة ، يتم تنشيط السلوك العدائي عن طريق حقن خطأ.

لتحقيق هدفنا ، تم اقتراح نهجين متكاملين ، كل واحد يستهدف مشكلة معينة. الأول هو نهج بناء تسلسل الرمز ، الذي يربط بين حالتين معينتين من الذاكرة. وهو يستند إلى الأسس النظرية المتعلقة بمجال مشكلات حل القيود. الطريقة الثانية تتعامل مع آلية إلغاء تزامن الكود التي تسمح بإخفاء كود معين عن طريق إجراء تحويلات عليه. تنفيذ كلا النهجين اسفر عن تطويرأداتين يمكنها إنشاء حلول تلقائيًا. قدمت أمثلة التطبيق ودراسة حالة الاستخدامات الممكنة للنُهج المقترحة لتنفيذ العمليات التي تعرض أمن البطاقة الذكية للخطر.

الكلمات المفتاحية: عنصر آمن ، جافا كاردة, مزامنة الكود ، حقن الخطأ ، حل القيد

Résumé

Les éléments sécurisés ont gagné une grande place dans notre vie quotidienne. Ils existent sous plusieurs formes. La carte à puce est l'élément le plus représentatif de la famille des éléments sécurisés. Elle est considérée comme étant un support d'exécution d'applications et de stockage d'informations très sécurisé. Vu la nature des informations qu'elles détiennent, les cartes à puce sont devenues la cible des personnes malintentionnées qui veulent s'approprier des informations sensibles qui y sont stockées voir même prendre le contrôle du système. La sécurité d'une carte à puce peut être contournée par des attaques matérielles, logicielles ou combinées. C'est dans cette dernière catégorie que s'inscrit notre travail.

Notre objectif dans cette thèse est de développer un nouveau vecteur d'attaque. En effet, c'est en maîtrisant les détails permettant de contourner la sécurité de la carte que nous pourrions par la suite trouver les contre-mesures permettant de s'en prémunir : « La meilleure défense c'est l'attaque ». La plateforme Java Card étant la plus utilisée, elle est retenue comme notre plateforme cible. Le but est de trouver une méthodologie de construction de codes malveillants activables par attaque en faute. L'idée est de cacher ce code malveillant dans un autre code sain (par construction) afin qu'il puisse être chargé dans la carte sans qu'il ne soit détecté par les mécanismes de sécurité embarqués ou une analyse du code. Une fois sur la carte, le comportement hostile est activé moyennant une injection de faute.

Pour aboutir à notre objectif, nous avons proposé deux approches complémentaires répondant chacune à un problème particulier. La première est une approche de construction de séquence de code, reliant deux états mémoire donnés, par parcours d'arbre. Elle repose sur des fondements théoriques liés au domaine des CSPs (Constraint Satisfaction Problem). La seconde approche traite le mécanisme de désynchronisation de code qui permet la dissimulation d'un code donné en opérant des transformations dessus. La mise en œuvre des deux approches a donné lieu à deux outils pouvant générer des solutions de façon automatique. Des exemples d'application et une étude de cas ont permis de présenter des exploitations possibles des approches proposées afin de réaliser des opérations mettant en danger la sécurité d'une carte à puce.

Mots-clés : élément sécurisé, Java Card, désynchronisation de code, injection de faute, satisfaction de contraintes, construction de code.

Abstract

Secure elements take place in an important part of our day to day life, they exist in several forms. The smart card is the most representative element in the family of secure elements. It is considered to be a very secured device designed to execute applications and store confidential data. Due to the nature of the information they hold, smart cards have become the target of malicious persons who want to appropriate sensitive stored information or even take control of the system. The security of a smart card can be bypassed by hardware, software or combined attacks. Our work falls in this last category.

Our objective in this thesis is to develop a new vector of attack. Indeed, it is by mastering the details that allow us to bypass the security of the card that we will then be able to find the adequate countermeasures : « The best defence is the attack ». Java Card, being the most used platform, is retained as our target platform. Our goal is to find a methodology for building malicious code that can be activated by a fault attack. The idea is to hide this malicious code into another innocent code (by construction) so that it can be loaded into the card without being detected by embedded security mechanisms or code analysis. Once on the card, the hostile behaviour is activated through a fault injection.

To achieve our objective, we proposed two complementary approaches, each responding to a specific problem. The first one is an approach for code sequence construction, linking two given memory states, by tree traversal. It is based on theoretical foundations related to the field of CSP (Constraint Satisfaction Problem). The second approach deals with the code desynchronization mechanism that allows the hiding of a given code by performing transformations on it. The implementation of the two approaches gives rise to two tools that can generate solutions automatically. Application examples and a case study presented possible ways in which the proposed approaches could be used to carry out operations that would compromise the security of a smart card.

Keywords : secure element, Java Card, code desynchronization, fault injection, constraints satisfaction, code construction.

Table des matières

Liste des Figures

Liste des Tableaux

Liste des Abréviations

AES	Advanced Encryption Standard
AID	Application IDentifer
APDU	Application Protocol Data Unit
API	Application Programming Interface
ATR	Answer To Reset
BCV	ByteCode Verifier
CAD	Card Acceptance Device
CAP	Converted APplet
CISC	Complex Instruction Set Computer
COS	Card/Chip Operating System
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DES	Data Encryption Standard
DRM	Digital Rights Management
EEPROM	Electrecally, Erasable, Programmable Read Only Memory
FPGA	Field Programmable Gate Arrays
GFC	Graphe de Flot de Contrôle
GSM	Global System for Mobile Communications
ID	Instruction de Désynchronisation
IoT	Internet of Things
ISO	International Standards Organization
JCA	Java Card Assembly
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JVM	Java Virtual Machine
NVM	Non Volatile Memory

PIN	Personal Identification Number
RAM	Random Access Memory
RID	Ressource IDentifier
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
ROM	Read Only Memory
RSA	Rivest Shamir Adleman
SE	Secure Element
SIM	Subscriber Identity Module
SW	Status Word
TAPDU	Transport Application Protocol Data Unit

Introduction

Contexte et problématique

Le code embarqué est présent dans des milliards d'appareils partout dans le monde. Une grande partie de ces dispositifs traite des informations sensibles à caractère confidentiel. L'exemple le plus utilisé est la carte à puce qui est présente dans la vie de tous les jours (identité, mobile, banque, santé, transport, etc.). Le niveau de sécurité élevé qu'elles offrent est à la base de l'instauration de la confiance nécessaire aux utilisateurs finaux. Vu la nature des informations qu'elles manipulent, elles sont devenues la cible d'attaques. Ces dernières sont menées par des personnes malveillantes qui veulent contourner les mécanismes de sécurité embarqués dans la carte pour s'approprier des données qu'elles détiennent voire même prendre le contrôle du système à travers un accès non autorisé.

Ces attaques peuvent toucher la partie physique de la carte (attaques matérielles), ou la partie logicielle pour détecter des failles et contourner les mécanismes de sécurité (attaques logicielles). Et plus récemment, avec les travaux de Barbu [BTG10], est apparue une nouvelle catégorie d'attaques dites combinées. L'idée est de perturber l'environnement d'exécution de l'application en injectant une faute matérielle (par exemple : une injection laser, une impulsion électromagnétique, etc.) suite à laquelle cette application change de comportement (un effet logiciel de la faute). Une telle application est appelée : *mutant*. Par la suite, plusieurs travaux se sont succédés pour présenter différentes attaques combinées [VF10, BICL11, BL15, MML17].

Les attaques combinées utilisaient un code donné et tentaient de le muter de telle sorte à contourner les mécanismes de sécurité. Dans les cas extrêmes il a été montré que de tels programmes pouvaient muter et se transformer en programmes hostiles pouvant mettre en danger les programmes de sécurité résidant sur cette même plateforme voir même la plateforme elle-même. Partant de ce point, la question soulevée dans cette thèse est la suivante : « Serait-il possible de concevoir un code, à charger dans la carte, de telle sorte à ce qu'il devienne intentionnellement hostile suite à une injection de faute bien ciblée ? »
Autrement dit, nous cherchons à définir un nouveau vecteur d'attaques en raisonnant à partir d'un autre angle : « quelle est la capacité d'un développeur malintentionné de concevoir un code ayant deux sémantiques correctes : avant et après mutation (i.e. injection de faute) ». Ceci dit, en maîtrisant les détails d'un vecteur d'attaque nous pourrions par la suite comprendre ses limites et ainsi développer les contre-mesures et outils d'analyse nécessaires permettant de s'assurer que de tels évènements ne pourraient pas compromettre la sécurité de la plateforme.

Motivations et objectifs du travail

Comme souligné précédemment, les travaux de recherche présentés dans le cadre de cette thèse s'inscrivent dans le domaine de la sécurité des éléments sécurisés et plus spécialement les cartes à puce. La plateforme Java Card étant la plus répandue dans ce domaine , elle est retenue comme étant notre plateforme cible.

A priori, cacher un code hostile dans un autre code sain pourrait être considéré comme une technique d'obfuscation de code qui vise à dissimuler une charge utile (*payload*) dans un code régulier. En effet, l'obfuscation de programmes est une méthode utilisée pour la protection de la propriété intellectuelle des logiciels. Cependant, elle est largement utilisée aussi par les développeurs de logiciels malveillants pour cacher leur code hostile et éviter ainsi d'être détectés. Elle vise à transformer un code dans une nouvelle version (i.e. une nouvelle structure) sémantiquement équivalente à la version originale, mais plus difficile à comprendre et à analyser.

Notre travail vise à dissimuler un code hostile pour en faire la compréhension et l'analyse plus difficiles. Ce qui représente le même objectif que celui des techniques d'obfuscation. Néanmoins, la différence est que l'obfuscation préserve la sémantique alors que dans notre cas nous voulons changer même la sémantique du code, chose qui est encore plus difficile et délicate vu que nous ciblons une plateforme très contrainte (Java Card est basée sur le langage Java qui est un langage fortement typé). Ceci dit, le défi et l'originalité de notre travail repose sur la capacité d'avoir un même code possédant au moins deux sémantiques différentes et correctes vis-à-vis de la spécification de la machine virtuelle Java Card [Ora15b], i.e. un code polymorphe. Initialement, le comportement hostile (sémantique 1) est dissimulé dans un autre code (sémantique 2) pour être récupéré après l'injection d'une faute ciblée.

L'idée est de camoufler le code hostile en construisant du code autour afin qu'il change sa structure et sa sémantique (i.e. la séquence hostile originale n'apparait plus telle quelle dans le code construit). Cette construction de code revient à rajouter des instructions de telle façon à ce que le début du code à cacher soit fondu dans le reste du programme (nous n'aurions plus les mêmes instructions). A ce niveau, deux points principaux doivent être traités :

- Le rajout des instructions doit se faire tout en respectant un ensemble de contraintes issues de la spécification JCVM garantissant à tout instant la correction syntaxique et sémantique du code construit.
- Cette transformation de code, i.e. le rajout des instructions, engendre un effet sur son interprétation : un décalage que nous avons appelé désynchronisation de code qu'il faudra gérer pour s'assurer de l'atteinte du résultat désiré.

Vu la difficulté de chacun de ces deux sous-problèmes, et l'effort nécessaire pour les traiter en profondeur afin de maîtriser le problème principal de notre travail, nous avons jugé plus judicieux d'étudier chaque point à part dans l'optique d'avoir à la fin une solution mettant en jeu les deux.

Les principales contributions réalisées dans le cadre de cette thèse, afin de répondre aux problèmes soulevés précédemment, peuvent être résumées comme suit :

La construction de séquences de code par parcours d'arbre. Nous avons montré que ce problème se ramène à un problème de résolution de contraintes. En se basant sur des fondements théoriques du domaine des CSPs (Constraint Satisfaction Problem), nous avons proposé une modélisation du problème. Par la suite, une approche pour la construction des séquences de code basée sur un

parcours d'arbre a été proposée. De plus, nous avons exposé une optimisation de cette approche en se basant sur des heuristiques que nous avons défini.

La désynchronisation de code. Nous avons commencé par définir qu'est ce qu'un modèle de désynchronisation. Ensuite, suivant un modèle spécifique, nous avons proposé une approche traitant ce mécanisme en détails : le cas général et tous les cas problématiques qui en découlent avec les traitements correspondants.

Mise en place d'outils automatiques pour la génération des solutions. La mise en œuvre des deux approches proposées a donné lieu à deux outils qui visent l'automatisation de ces dernières. Le premier outil, nommé *Trace Generator* permet de générer toutes les séquences de code possibles pouvant lier deux état mémoire donnés (et donc deux fragment de codes) tout en respectant la spécification. Le second outil quant à lui permet de calculer tous les cas possibles de la désynchronisation d'un code donné dans le but de le dissimuler. De plus, des exploitations possibles de nos approches sont illustrées à travers des exemples d'application.

Les travaux présentés dans le cadre de cette thèse ont fait l'objet de deux publications internationales [HL13, HLM19] et de trois communications [HMGL14, HLM14, Ham14].

Organisation du travail

Le présent document s'articule autours de six chapitres regroupés dans deux parties. L'agencement des chapitres en fonction de la progression du travail est donné dans la figure 1.

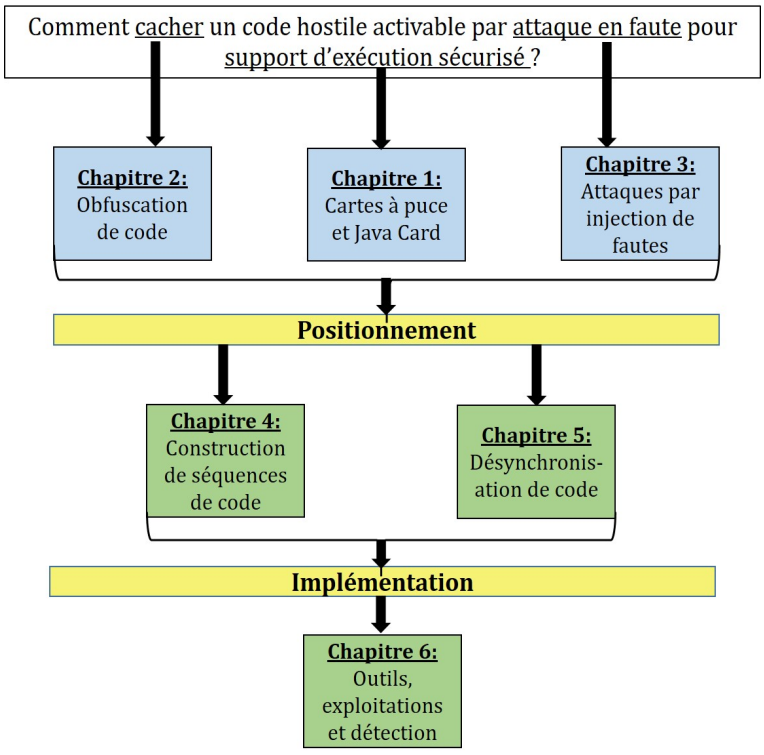

Fig 1 – Plan de rédaction de la thèse

En raison de la diversité des domaines liés à notre travail, la première partie réunit trois cha-

pitres. Chacun d'entre eux traite d'un domaine d'étude qui est relatif à un mot-clé figurant dans notre question de départ : *« Comment cacher un code hostile activable par attaque en faute pour support d'exécution sécurisé »*.

Dans le ***chapitre1***, nous avons commencé par introduire les éléments sécurisés et plus spécialement le domaine des cartes à puce en présentant les différents concepts de base (typologie, standards, protocoles, etc). Par la suite, nous avons présenté la plateforme Java Card , plateforme retenue pour notre travail, en mettant l'accent sur l'aspect sécurité (attaques et mécanismes de défense).
Et comme notre objectif principal est de trouver comment dissimuler un code hostile, nous avons réservé le ***chapitre2*** pour une présentation des techniques d'obfuscation de code existantes et plus spécialement celles traitant les codes malveillants.
Le ***chapitre3*** traite quant à lui les attaques par injection de faute, le moyen d'activation de nos codes hostiles, en présentant les détails qui leur sont relatifs (techniques d'injection, modèles de fautes, *etc*).
L'accent est mis sur les attaques activables par attaques en faute existantes (attaques combinées).
Cette partie est clôturée par le positionnement de notre travail par rapport à ce qui existe dans les domaines présentés dans les trois chapitres précédents. Les hypothèses de travail sont posées, le problème principal est décomposé en deux sous-problèmes qui sont traités dans la seconde partie.

Ainsi cette première partie englobe les briques de base nécessaires pour aborder la seconde partie qui est consacrée, quant à elle, à nos contributions pour répondre à la problématique posée. Elle est scindée en trois chapitres.
Dans le ***chapitre4***, après avoir donné un bref aperçu sur le domaine des CSPs, nous avons exposé notre approche de construction de séquences de code en détaillant la modélisation du problème ainsi que sa résolution. Par la suite, une optimisation de cette approche, basée sur des heuristiques, a été présentée.
Le ***chapitre5*** présente notre deuxième approche qui concerne l'étude du mécanisme de désynchronisation de code (modèle de désynchronisation, cas général, cas problématiques) que nous avons défini pour répondre au seconde sous-problème.
Enfin, le ***chapitre6*** concerne la mise en pratique de nos deux approches, en présentant les détails des outils développés qui permettent leur automatisation. De plus, des exemples d'application et une étude de cas sont exposés afin de montrer des exploitations possibles. A la fin, nous présenterons quelques outils existants pour la détection des vulnérabilités des injections de fautes, leurs limites ainsi que des pistes de solutions possibles.

Pour finir, une conclusion générale reprendra nos contributions à travers ce travail et ainsi que quelques perspectives de recherche.

Première partie

Contexte de l'étude et état de l'art

CHAPITRE 1

Les cartes à puce et la plateforme Java Card

Sommaire

Les éléments sécurisés, comme leur nom l'indique, sont des supports sécurisés pour la manipulation et le stockage des données, généralement de nature sensible. Ils ont gagné une grande place dans notre vie quotidienne. Eurosmart [1] annonce dans son bulletin de presse que le nombre d'éléments sécurisés vendus en 2018 a dépassé les 10 milliards d'unités avec des prévisions pour atteindre les 10.3 milliards d'unités en 2019 [Eur18].

La carte à puce est un élément des plus représentatifs de la famille des éléments sécurisés. Elle se caractérise par des contraintes matérielles fortes. Les cartes à puce sont usuellement pourvues de processeurs de faible puissance, de capacités d'entrées/sorties limitées, de mémoires de petite taille (de quelques Kilos octet seulement). Mais le niveau élevé de sécurité qu'elles offrent ainsi que leur usage intuitif par leur porteur en font d'elles l'un des objets mobiles de large utilisation.

1. Association internationale représentant les professionnels de l'industrie des cartes à puce

Comme la carte à puce constitue le support d'exécution cible de notre travail, elle fera l'objet de la première partie (section 1.1) de ce chapitre en présentant les différents aspects qui y sont liés. Par la suite, nous aborderons dans la seconde partie du présent chapitre (section 1.2), les concepts liés à notre plateforme cible, à savoir Java Card , dans le but d'introduire les éléments nécessaires pour la compréhension de la suite du document. La fin du chapitre (section 1.3) donnera un aperçu général sur les différentes attaques contre les cartes à puce visant à remettre en cause leur sécurité. Ceci, dans le but de situer, par la suite, notre travail dans ce contexte.

1.1 La carte à puce comme élément sécurisé

1.1.1 Qu'est ce qu'un élément sécurisé ?

Un élément sécurisé (ou SE pour Secure Element en anglais) est une plateforme résistante qui permet de stocker des données de façon sécurisée et exécuter des fonctions cryptographiques au moyen de coprocesseurs qui implémentent des algorithmes communs tels que RSA, DES, AES [Raz16]. Les SE offrent un environnement hautement sécurisé qui permet de protéger les informations d'identité de l'utilisateur. Par exemple, dans le secteur financier ils sont utilisés pour héberger des applications de cartes personnalisées ainsi que les clés cryptographiques nécessaires pour effectuer des transactions financières. Un autre exemple est l'utilisation des SE pour sauvegarder des données biométriques ou des certificats numériques dans le but de les utiliser pour la signature des documents.

Un élément sécurisé peut exister sous différentes formes : élément sécurisé embarqué, carte UICC [2], carte microSD ainsi que les cartes à puce. D'après [Glo18], les SE sont une évolution de la puce traditionnelle qui réside dans les cartes à puce. Ils existent sous de nouvelles formes afin de prendre en charge les exigences de différentes implémentations commerciales et les besoins du marché (smartphones, tablettes, boîtiers décodeurs, voitures connectées et autres objets (IoT)).

1.1.2 Qu'est ce qu'une carte à puce ?

La carte à puce est un support électronique, portable et sécurisé pour l'exécution d'applications et le stockage d'informations sensibles. Physiquement, c'est une carte en plastique de quelques centimètres de côté et moins d'un millimètre d'épaisseur, incorporant un circuit électronique (la puce) capable de stocker et traiter des informations de façon très sécurisée. Malgré les ressources limitées d'une carte à puce, en termes d'espace mémoire et capacité de calcul, dans ses dernières versions elle est capable d'exécuter du code et d'héberger diverses applications se rapprochant ainsi d'un ordinateur personnel.

1.1.3 Historique

La carte à puce actuelle a connu une grande évolution et un passage à l'échelle qui fait d'elle un support électronique à la pointe de la technologie. L'idée de base était de regrouper toutes les fonctionnalités sur un seul circuit électronique tout en ayant comme priorité la protection des données qui sont manipulées. Le tableau suivant (tableau 1.1) résume les dates les plus importantes dans l'histoire de l'évolution des cartes à puce.

2. UICC pour Universal Subscriber Identity Module

Année	Évènement
1968	* Les deux Allemands "Jürgen Dethloff" et "Helmut Gröltrup" introduisent l'idée d'incorporer un circuit intégré dans une carte en plastique.
1970	* Au Japon, "Kunitaka Arimura" dépose un brevet sur la carte à puce.
1974-1979	* Le Français "Roland Moreno" dépose 47 brevets (dans 11 pays) sur la carte à puce. Son invention a été distinguée des autres travaux par le fait qu'elle incorpore des moyens de protection (matériels et/ou logiciels) de la mémoire pour restreindre l'accès en lecture et en écriture.
1977	En Allemagne, "Dethloff" dépose un brevet où il introduit un microprocesseur comme moyen de protection de la carte à mémoire.
1979	* En France, le groupe "Bull" crée la première carte programmable à microprocesseur (nommée CP8).
1983-1984	* Arrivée des premières cartes téléphoniques et cartes bancaires en France et en Allemagne .
1987	* Introduction des premières normes ISO, en Europe, régissant le domaine des cartes à puce.
1991	* Lancement du réseau GSM et des premières cartes SIM (Subscriber Identity Module).
1994	* Lancement de la première carte à puce sans contact (carte MIFARE de la société Mikron).
1997	* Apparition des cartes multi-applicatives.

Tab 1.1 – Historique de la carte à puce (extrait de [Ham12])

1.1.4 Typologie des cartes à puce

Les différents types des cartes à puce existants peuvent être classés en fonction de leur architecture interne ou leur mode de communication avec le monde extérieur. La figure 1.1 résume cette classification.

1.1.4.1 Selon la technologie interne

Carte à mémoire. Elle constitue la catégorie la plus ancienne des cartes. Comme son nom l'indique, elle ne contient que de la mémoire (pas de microprocesseur). De ce fait, elle est limitée à des applications relativement simples. Une carte à mémoire peut être [Tav07] : *simple* ne contenant qu'une zone mémoire et le minimum de logique nécessaire pour pouvoir y accéder ; ou *protégée* associant de la mémoire à la logique permettant l'exécution d'automates simples allant jusqu'à la présentation de mots de passe (par exemple, les anciennes télécartes contenant un compteur d'unités, un numéro de série et un code d'authentification).

Carte à microprocesseur. Elle est dite "intelligente" (*Smart Card* en Anglais) car en plus de sa mémoire elle englobe un microcontrôleur complet qui se présente sous la forme d'un rectangle de silicium dont la surface est inférieure à $25mm^2$. Pratiquement, c'est l'association, en un seul circuit (Figure 1.2), des composants suivants [Tav07, Bar12] :

— Un *microprocesseur* : de 8, 16 ou 32 bits avec des architectures CISC (Complex Instruction Set Computer) ou RISC (Reduced Instruction Set Computer) travaillant à des fréquences internes allant de 5 à 40 MHZ.

— Un coprocesseur cryptographique qui réalise les opérations de chiffrement et déchiffrement. Ce coprocesseur cryptographique est associé à un générateur de nombres aléatoires RNG (Random Number Generator).

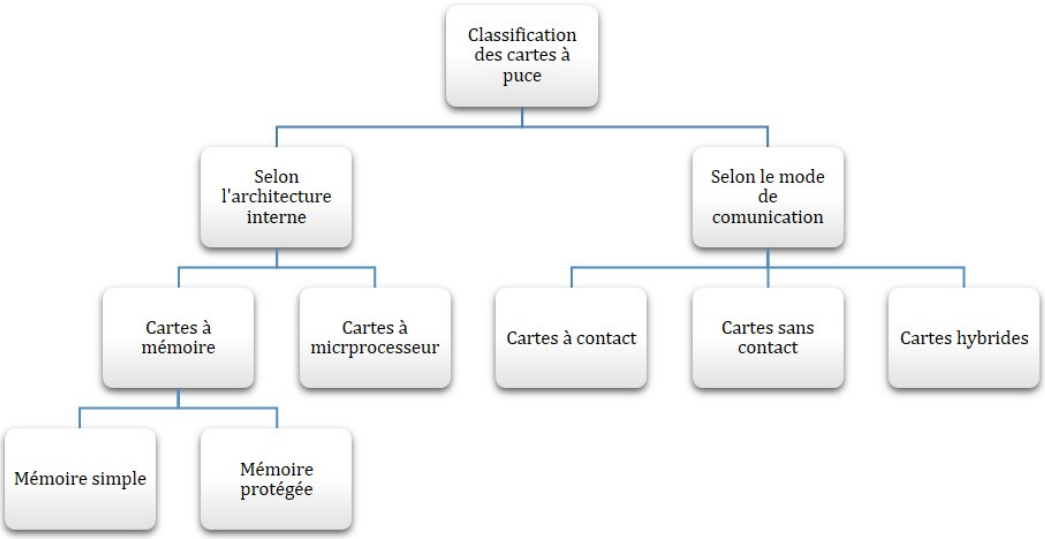

Fig 1.1 – Typologie des cartes à puce

— Une *mémoire morte (ROM)* : C'est une mémoire persistante. Son contenu (système d'exploitation ainsi que les données permanentes) n'est pas modifiable. Sa taille varie de 32 Ko jusqu'à 256 Ko et même plus pour les cartes haut de gamme.
— Une *mémoire vive (RAM)* : C'est une mémoire volatile. Elle est utilisée comme espace temporaire pour modifier et stocker les données. Sa taille varie de 1 Ko à 24 Ko (pour les cartes haut de gamme).
— Une *mémoire non volatile NVM* : C'est une mémoire persistante comme la ROM, mais son contenu peut être modifiable. Elle contient les données qui peuvent évoluer dans le temps sans être perdues. Sa taille varie de 16 Ko à 256 Ko (pour les cartes haut de gamme)
— Une *interface d'entrée/sortie série* : Pour les échanges de données.
— La logique nécessaire pour faire fonctionner l'ensemble .

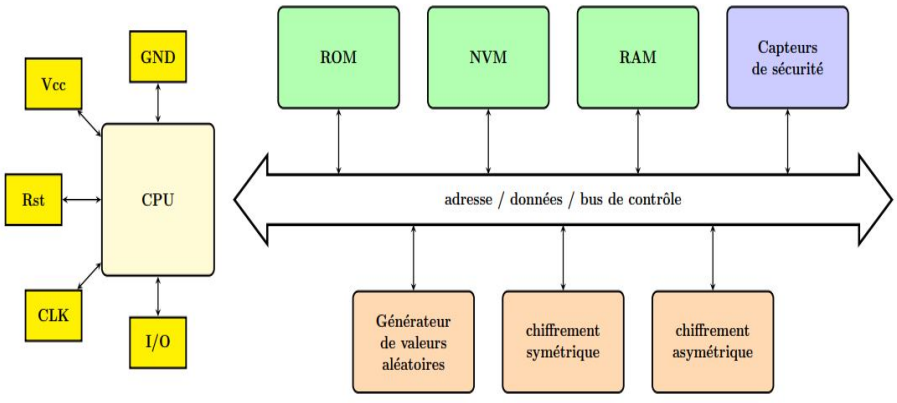

Fig 1.2 – Architecture interne générale d'une carte à microprocesseur (adaptée de [Bou14])

Les cartes à microprocesseur conviennent aux applications les plus sensibles où le degré de sécurité des données est le facteur prédominant : contrôle d'accès sécurisé, carte bancaire, télécommunication, *etc.* Dans tout le reste du document, l'utilisation du terme *"carte à puce"* fera référence à ce type de

cartes i.e. les cartes à microprocesseur.

1.1.4.2 Selon le mode de communication

Carte à contact. Afin de communiquer avec le monde extérieur, elle doit être insérée dans un lecteur (appelé aussi CAD pour Card Acceptance Device). Ceci est assuré par les points de contact présents sur sa surface via une liaison en série. Les contacts (figure 1.3) sont en nombre de huit (C1-C8). Leurs caractéristiques sont définies dans la norme ISO 7816-2, où :

- C1 (Vcc) : tension d'alimentation positive de la carte
- C2 (RST) : commande de remise à zéro
- C3 (CLK) : horloge fournie à la carte
- C5 (GND) : masse électrique
- C6 (Vss) : tension de programmation (n'est plus utilisée)
- C7 (I/O) : entrée/sortie de données (bidirectionnelle)
- C4 et C8 (RFU) : à l'origine, contacts réservés pour utilisation future mais actuellement ils servent à communiquer en USB (Universal Serial Bus)

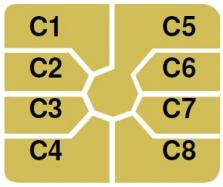

Fig 1.3 – Numérotation et position des contacts selon la nome ISO 7816-2

Carte sans contact La communication s'établit à travers une interface radio fonctionnant par induction grâce à une antenne imprimée ou intégrée dans la carte à puce. Plusieurs technologies existent, mais de façon générale, la carte sans contact contient une puce électronique avec un émetteur hyperfréquence et une antenne intégrée dans le plastique (figure 1.4). Cette carte est un peu plus complexe car elle doit intégrer un régulateur de tension, un modulateur/démodulateur ainsi qu'un mécanisme d'anti-collision, un générateur d'horloge et bien entendu une antenne [Lan06]. Les cartes sans contact sont privilégiées dans le domaine du transport ainsi que pour le contrôle d'accès aux bâtiments, domaines dans lesquels les transactions doivent être faites à une vitesse assez élevée.

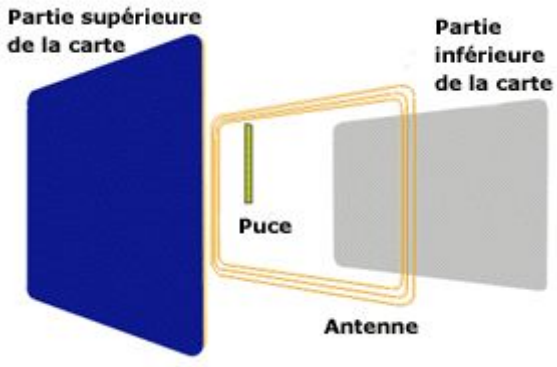

Fig 1.4 – Carte à puce sans contact

Carte hybride. Elle embarque deux puces, chacune a sa technologie. La première est reliée aux contacts, la deuxième à l'antenne (figure 1.5). Ces cartes sont le meilleur compromis car elles offrent les avantages des deux types de carte à puce mais leur prix est beaucoup plus élevé.

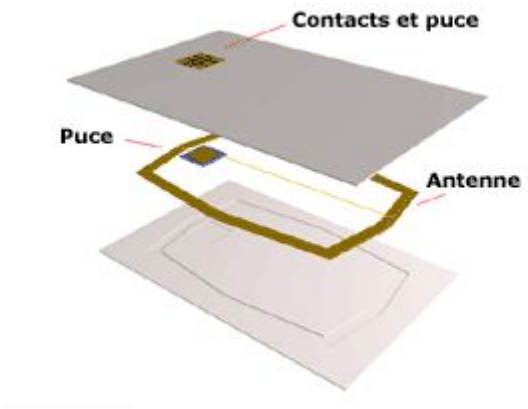

Fig 1.5 – Carte à puce hybride

1.1.5 Systèmes d'exploitation pour cartes à puce

Les cartes à microprocesseur sont dotées d'un système d'exploitation appelé "COS" (*Chip Operating System* ou *Card Operating System*). Cependant, vu les ressources limitées des cartes à puce, leur système d'exploitation doit être le plus léger possible, tout en étant riche, afin qu'il puisse être embarqué dans la carte. Trois sous-familles de cartes peuvent être distinguées [Tav07, Ham12] :

1.1.5.1 Les cartes à puce spécifiques

Dans ces cartes, le contenu de la majeur partie du système d'exploitation est défini par leur fabriquant. Il définit ses propres instructions et fichiers avec le contenu de son choix. Il s'agit de cartes "sur mesure" qui sont réservées aux grands groupes industriels, commerciaux ou financiers puisque cette approche impose de produire la carte en très grande série afin d'amortir les coûts d'étude et de développement qui sont très élevés.

1.1.5.2 Les cartes à puce personnalisables

Elles contiennent un système d'exploitation non modifiable, programmé par le fabriquant. Lorsqu'elles sont fournies par le fabriquant, certaines parties de la mémoire sont accessibles afin de définir le comportement global de la carte, les propriétés des fichiers, *etc.* C'est le stade de la *"personnalisation"* i.e. adaptation aux différentes applications pouvant être visées. Une fois cette phase de personnalisation achevée, il y a une opération de "verrouillage" du contenu de la carte afin de rendre toute modification impossible par la suite.

1.1.5.3 Les cartes à puce à système ouvert

Pour ces cartes, le développement des applications est plus simple car il se fait en langage évolué (Java, Basic, C) et non pas en langage machine. Ainsi le programme obtenu est traduit en un code intermédiaire qui sera chargé dans la carte puis exécuté par le microcontrôleur qui est doté d'un interpréteur du code intermédiaire. Dans un tel schéma, l'évolution du code est désormais possible (à l'inverse des deux types de système précédents où ce programme contenait à la fois le système opératoire

et l'application rendant ainsi toute évolution du code difficile voire impossible). Ceci revient à dire que les cartes à système ouvert autorisent le chargement des applications après délivrance de la carte et peuvent même héberger plusieurs applications différentes (dans le cas des cartes *multi-applicatives*). Il existe sur le marché plusieurs systèmes ouverts, parmi lesquels nous présentons ci-dessous : MULTOS, Basic Card et Java Card.

MULTOS [3]**.** C'est un système d'exploitation mlti-applicatif. Il a été proposé initialement par Mondex et MasterCard pour le paiement électronique. MULTOS s'exécute sur le microcontrôleur de la carte à puce. A chaque fois qu'une application est envoyée vers cette dernière, il vérifie sa validité moyennant divers systèmes sécuritaires prévus pour cette fin. Ensuite, par le biais de pare-feux spéciaux, il assure l'isolation des applications résidant dans la même carte (ceci par l'allocation de zones mémoires protégées pour chacune d'elles). Les cartes MULTOS se programment au moyen d'un langage spécifique appelé MEL (MULTOS Executable Language). Cependant, dans le but de rendre le système plus ouvert, des applications développées en C ou en Java peuvent être traduites en MEL.

Basic Card [4]**.** C'est une carte à système ouvert, proposée par la société allemande ZeitControl, facilement programmable en langage Basic. Comme elle intègre plusieurs solutions cryptographiques, selon les versions des cartes, elle offre un niveau élevé de sécurité. L'outil de développement des applications pour ces cartes est gratuit. Il intègre un simulateur de carte permettant de tester l'application développée sans avoir besoin de la carte ni du lecteur. Cependant, la Basic Card n'est pas supportée par les fabricants de carte.

Java Card [5]**.** C'est une carte multi-applicative sur laquelle il est possible de charger et d'exécuter des programmes écrits en Java. Et comme cette plateforme constitue la cible du présent travail, toute la section 1.2 est dédiée pour y donner une présentation détaillée.

1.1.6 Communication de la carte à puce avec son environnement

1.1.6.1 Standardisation

Les cartes à puces sont très standardisées car elles doivent être utilisables avec la gamme la plus large possible de lecteurs dans le monde entier. La normalisation concerne essentiellement les paramètres physiques, électriques et logiciels. Les normes internationales les plus utilisées pour les cartes à puce sont celles appartenant au standard ISO. Comme exemple, nous citons la famille des normes ISO 7816 relatives aux cartes à puce avec contact :

- ISO 7816-1 : Caractéristiques physiques
- ISO 7816-2 : Dimension et position des contacts
- ISO 7816-3 : Signaux électroniques et protocoles de transmission
- ISO 7816-4 : Commandes inter-industries
- ISO 7816-5 : Enregistrement des fournisseurs d'applications
- ISO 7816-8 : Commandes pour des opérations de sécurité
- ISO 7816-9 : Commandes pour la gestion des cartes
- ISO 7816-10 : Signaux électroniques et réponse à la mise à zéro des cartes synchrones
- ISO 7816-11 : Vérifications personnelles par méthodes biométriques
- ISO 7816-15 : Application des informations cryptographiques

3. `http://www.multos.com`
4. `http://www.basiccard.com`
5. `https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html`

En plus de la large gamme des normes de la famille ISO, il existe d'autres standards relatifs à des domaines d'application spécifiques, par exemple : GSM (Global System for Mobile Communications) pour la téléphonie mobile, EMV défini par Europay, Mastercard et Visa pour l'industrie bancaire.

1.1.6.2 Modèle de communication

Pour que la carte à puce communique avec le monde extérieur, elle doit être placée dans un lecteur (carte avec contact) ou à proximité de ce dernier (carte sans contact). Le lecteur est connecté lui même à un autre ordinateur (hôte). La communication se fait à travers le contact C7 (I/O) en semi-duplex : les deux parties (carte/hôte) peuvent envoyer des données mais pas simultanément.

Les données circulant entre la carte et l'hôte sont des paquets de données spéciaux appelés APDUs *(Application Protocol Data Units)* défini dans la norme ISO 7816-4. Chaque APDU est soit :

- Une **commande APDU** : qui spécifie une requête. Sa structure (figure 1.6) comprend :
  - Un en-tête obligatoire comprenant de 4 octets :
    - **CLA** : indique la catégorie de la commande APDU (correspondant au domaine d'application)
    - **INS** : indique l'instruction à exécuter
    - **P1, P2** : les paramètres de l'instruction
  - Un corps de longueur variable et qui comprend :
    - **LC** : la taille du champ de données
    - **Données** : contient les données à envoyer à la carte pour exécuter l'instruction définie dans l'en-tête (INS)
    - **Le** : correspond au nombre d'octets attendus pour le champ "données" de la réponse

Entête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Champ de données	Le

Fig 1.6 – APDU de commande

- Une **réponse APDU** : retourne le résultat d'exécution d'une commande. Sa structure (figure 1.7) comprend :
  - Un corps optionnel contenant les **données** à envoyer par la carte et dont la taille est déterminée par le champs "Le" de la commande.
  - Une zone de fin obligatoire :
    - **SW1** et **SW2** (Status Word) : c'est un code sur deux octets indiquant l'état de la carte après exécution de la commande.

Corps optionnel	En-queue obligatoire	
Champ de données	SW1	SW2

Fig 1.7 – APDU de réponse

Le transport des APDUs est assuré par le protocole TAPDU. Il est défini par la norme ISO 7816-3. Les deux principaux modes de transmission par ce protocole sont :

- T=0 qui utilise une transmission *"orientée octet"*
- T=1 qui utilise une transmission *"orientée paquet"*

Les échanges d'APDUs se font suivant un modèle client/serveur. Dans ce mode de fonctionnement, la carte joue le rôle passif (serveur), elle n'initialise pas la communication mais se contente de répondre aux commandes APDUs envoyées par l'hôte (client). Les commandes et réponses APDUs sont échangées alternativement entre la carte et l'hôte.

1.1.7 Domaines d'application

La carte à puce a désormais pris une grande place dans notre vie quotidienne, du fait qu'elle est utilisée dans plusieurs domaines où ses capacités de stockage sécurisé des informations sont mises en valeurs. Parmi ces applications nous pouvons citer :

- **Applications de paiement :** carte bancaire, carte pré-payée, porte-monnaie électronique (e-purse), titres de transport en commun, télévision à péage, *etc.*
- **Applications d'identification :** contrôle d'accès aux locaux, contrôle horaire, carte SIM, carte d'identité, passeport, carte de santé, *etc.*
- **Applications de sécurisation :** la carte est utilisée comme un support de stockage des clés cryptographiques servant à l'authentification et la sécurisation des communications (contrôle d'accès logique à un serveur par authentification, sécurité des transactions sensibles, *etc.*)

1.2 La plateforme Java Card

1.2.1 Présentation de Java Card

La plateforme Java Card est basée sur la technologie Java. Elle est destinée aux équipements fortement contraints, en termes de mémoire et de puissance de calcul, tels que les cartes à puces. En d'autres termes, une Java Card est une carte à puce sur laquelle on est capable de charger et d'exécuter des applications Java de type *applets*[6] ou *servlets* (à partir de sa version 3.0). Comme déjà indiqué dans la section 1.1.5.3, Java Card est une plateforme ouverte, i.e. que les programmes qui y sont contenus ne sont pas nécessairement fournis par le fabriquant de la carte et donc peuvent être chargés après la délivrance de celle-ci. La technologie Java Card peut être considérée comme étant une plateforme fournissant un environnement sécurisé pour cartes à puce, interopérable et multi-applicatif qui bénéficie des avantages du langage Java (facilité de programmation, indépendance du matériel et sécurité).

1.2.2 Évolution des versions

Les origines de Java Card remontent à 1996 où un groupe d'ingénieurs du centre de production de Schlumberger à Austin au Texas a cherché à simplifier le modèle de programmation des cartes à puce et d'obtenir un code portable indépendamment du fabriquant de la carte. Le langage de programmation Java apparaît alors comme étant la solution mais avec adaptation, vu la contrainte des ressources limitées des cartes, par l'utilisation d'un sous-ensemble du langage uniquement. Schlumberger devint alors la première entreprise à acquérir une licence en proposant la spécification Java Card 1.0 [Zhi00]. En Février 1997, le Java Card Forum[7] (JCF) a été co-fondé par Schlumberger, Bull et Gemplus. Le but de ce consortium industriel est d'identifier et de résoudre les problèmes de la technologie Java Card en proposant des spécifications ainsi que de promouvoir des APIs (Application Programming Interfaces) Java Card afin de permettre son adoption par l'industrie de la carte à puce [Zhi00].

6. Nous utiliserons le terme applet dans la suite du document pour désigner une application Java Card
7. `http://www.javacardforum.org`
En 2019, le JCF regroupe, en partenariat avec Oracle, les principaux fabricants de cartes à puce (Gemalto, G+D Mobile Security, IDEMIA, Infineon, NXP Semiconductors, STMicroelectronics, Secure MCU).

En novembre 1997, Sun Microsystems fournit la spécification 2.0 de Java Card qui consiste en un sous-ensemble du langage et de la machine virtuelle Java. Cette spécification définit les concepts de base de la programmation et des APIs très différents de ceux de la version de Schlumberger (version 1.0).
En mars 1999, sort la version 2.1 dont le changement le plus significatif est la définition explicite de la machine virtuelle Java Card (JCVM pour Java Card Virtual Machine) et du fichier CAP (format binaire représentant des fichiers *.class* convertis, voir section 1.2.3.1), permettant ainsi une vraie interopérabilité.
En juin 2002, la version 2.2 est publiée. Elle propose essentiellement quelques nouveaux algorithmes cryptographiques.
En octobre 2003, la version 2.2.1 est publiée. Elle corrige les APIs de la version précédente et apporte quelques clarifications.
En mars 2006, la version 2.2.2 est publiée. Elle n'apporte pas de changement majeur mis à part quelques nouveaux algorithmes cryptographiques et une clarification des APIs.
C'est en 2008, qu'arrive la version 3.0 qui introduit deux types de la spécification Java Card : la Java Card 3 *Classic Edition* (c'est juste une évolution de la Java Card 2.2.2) et la Java Card 3 *Connected Edition*, qui en plus des applets classiques, supporte un nouveau type d'applications : les *servlets*. Ces dernières sont des applications web nécessitant un serveur web embarqué dans la carte (il s'agit de cartes modernes haut de gamme). Chacune de ces deux éditions est compatible avec les applications développées pour les éditions précédentes (i.e. antérieures à l'édition 3.0)
En mai 2009, la version 3.0.1 classique est publiée. Elle apporte principalement des mises à jour des algorithmes cryptographiques.
En janvier 2010, Oracle Corporation rachète Sun Microsystems.
En juin 2015, la version 3.0.5 classique est publiée. Elle est basée sur la sécurité, la cryptographie optimisée et l'introduction de nouvelles APIs.
En janvier 2019, la dernière version 3.1 classique est publiée. Elle introduit de nouvelles APIs et des fonctions de cryptographie mises à jour pour répondre aux besoins de sécurité de l'Internet des objets (en anglais IoT pour Internet of Things).

Comme notre travail porte sur la plateforme Java Card 3 édition « *Classic* », dans la suite du document toute utilisation du terme Java Card ou présentation de la plateforme feront référence à cette version. Pour plus de détails sur l'édition « *connected* », le lecteur peut consulter la documentation officielle fournie par Oracle [8].

1.2.3 Architecture de la plateforme

L'architecture de Java Card est conçue de telle sorte à pouvoir s'adapter aux contraintes (ressources limitées) des cartes pour construire une carte Java tout en conservant suffisamment d'espace pour charger des applications.

La Java Card 3 édition « *Classic* » est basée sur l'évolution de la version 2.2.2 de la plateforme. Elle cible les cartes à faibles ressources qui supportent uniquement les applets comme modèle d'application. L'édition « *Classic* » adopte la même architecture (figure 1.8) de la plateforme Java Card que les versions précédentes (architecture utilisée depuis la version 2.1). Les changements apportés par cette version portent essentiellement sur le support des derniers algorithmes cryptographiques (standards de sécurité) ainsi que les standards régissant les communications sans contact.

Cette architecture est définie par les trois spécifications suivantes :

8. `https://www.oracle.com/technetwork/java/embedded/javacard/documentation/javacard-docs-1970421.html`

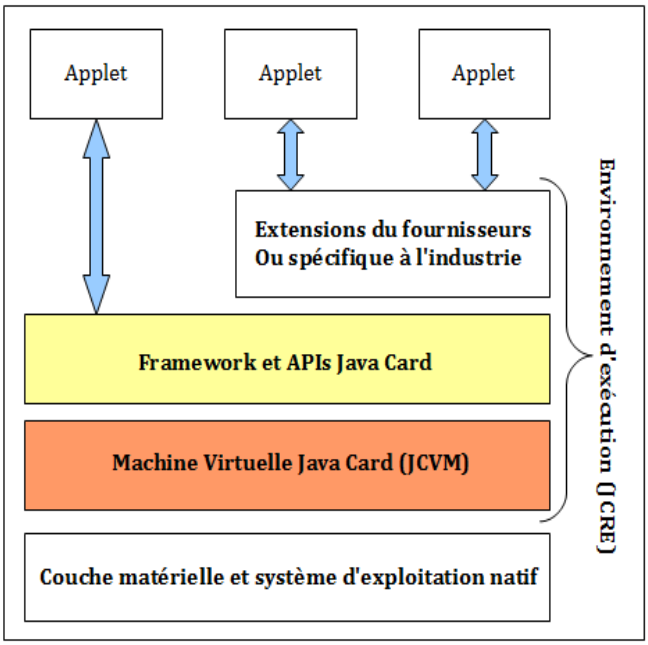

Fig 1.8 – Architecture de Java Card (adaptée de [Ort03])

— *Java Card 3 Virtual Machine Specification* [Ora15b]. Définit la machine virtuelle nécessaire pour les applications sur cartes à puce ;
— *Java Card 3 Runtime Environment Specification* [Ora15a]. Décrit précisément le comportement de l'exécution de la Java Card ;
— *la Java Card 3 API Specification* [Ora15c]. Décrit l'ensemble des paquetages et des classes Java nécessaires à la programmation des cartes à puce mais aussi quelques extensions optionnelles.

1.2.3.1 La machine virtuelle Java Card (JCVM)

La machine virtuelle Java Card (JCVM pour Java Card Virtual Machine) a une architecture (figure 1.9) semblable à celle d'une machine virtuelle Java (JVM pour Java Virtual Machine). Cependant, les ressources très restreintes des cartes à puce ont fait que la JCVM soit séparée en deux parties complémentaires :

— Une partie hors carte (*Off-Card*) : tourne sur une station de travail et comprend un vérifieur de bytecode (section 1.2.4.2) et un convertisseur effectuant des transformations sur ce code en vue de l'optimiser avant de le charger dans la carte ;
— Une partie sur carte (*On-Card*) : c'est la partie embarquée et elle comprend l'interpréteur de bytecode qui se charge de l'exécution du code chargé.

Le convertisseur pré-traite tous les fichiers *.class* contenus dans un paquetage, représentant l'unité de conversion du convertisseur, et les convertit en un fichier *.cap* (Converted APplet). En effet, le fichier Class n'est pas directement exécutable et doit subir une phase importante d'édition de liens. De ce constat est né le format de fichier CAP. C'est un format plus simple à exécuter pour une plate-forme disposant de peu de ressources [Lan06].

Les paragraphes suivants décrivent quelques éléments liés à JCVM, nécessaires pour la compréhension de la suite du document, à savoir : le langage bytecode, les structures de données d'exécution et le fichier CAP.

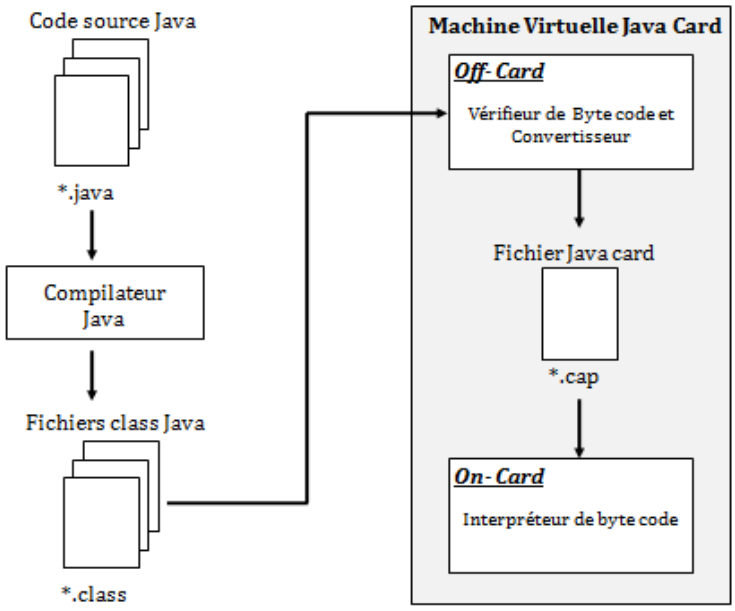

Fig 1.9 – La machine virtuelle Java Card (JCVM) (extraite de [Ham12])

Le langage bytecode

Le bytecode est un langage intermédiaire entre le code source et le code machine qui permet de rendre l'exécution des applications Java (bien entendu Java Card) multi-plateforme puisque il est indépendant de tout système d'exploitation. Le jeu d'instructions bytecode Java Card comprend 186 instructions. Elles sont définies dans le chapitre 7 de la spécification de la machine virtuelle [Ora15b]. Ce sont des opérations basiques qui combinées permettent de réaliser des traitements. Une instruction est composée d'un code opération (appelé `opcode`) suivi d'aucun, un ou plusieurs opérandes qui représentent les paramètres de l'instruction. Les instructions bytecode Java Card peuvent être classées sous plusieurs catégories en fonction du traitement qu'elles effectuent, par exemple :

- Opérations de manipulation de la pile d'opérandes : `aconst_null, bspush, aload_0, pop`, *etc.*
- Opérations arithmétiques : `sadd, iadd, ssub, isub, sinc`, *etc.*
- Opérations logiques : `sand, iand, sor, ior`, *etc.*
- Opérations d'appels et de retour de méthodes : `invokevirtual`, `invokespecial`, `return`, *etc.*

La classification complète des instructions bytecode Java Card est donnée dans l'annexe A.

Les structures de données d'exécution

Comme toute JVM, la JCVM a une architecture basée sur les piles (elle est dite *stack-based*). La pile d'exécution de la JCVM est utilisée pour stocker des données et des résultats intermédiaires. Cet espace mémoire contient un certain nombre d'éléments (sous-structures) qui sont nécessaires à l'interprétation des instructions bytecodes. Principalement :

- La **frame** qui permet de stocker toutes les informations nécessaires à l'exécution de la méthode en cours telles que les variables locales, les paramètres et valeurs d'appels de la méthode, les résultats des calculs intermédiaires, *etc.* Une nouvelle frame est créée à chaque fois qu'une méthode est invoquée, et est détruite à la fin de l'invocation.
- La **pile d'opérandes** qui sert lors des appels de méthodes pour passer les arguments et recevoir le résultat. Elle peut contenir aussi les résultats intermédiaires résultants des calculs

intermédiaires. De plus, certaines instructions bytecode prennent des valeurs de cette pile, effectuent des traitements sur ces valeurs, et mettent le résultat de l'opération sur la pile.

— Le tableau des **variables locales** qui stocke les paramètres et les variables locales de la méthode en cours. Ces variables sont accessibles via leur index. Ce dernier commence à partir de zéro.

Le fichier CAP

Le fichier CAP (Converted APplet) est un fichier binaire produit par le convertisseur dans la partie hors carte de la machine virtuelle Java Card après conversion des fichiers *.class* contenus dans un paquetage (figure 1.9). Chaque fichier CAP contient toutes les classes et interfaces définies dans un paquetage Java. C'est ce fichier qui sera chargé dans la carte, au lieu du ficher *.class*, car son format est simplifié.

Le fichier CAP est constitué d'un ensemble de composants, chacun d'entre eux décrit un ensemble d'éléments du paquetage en question ou un aspect du fichier CAP. Dans son ensemble, ce dernier est constitué de 12 composants dont 3 sont optionnels mais il y a une possibilité de définir de nouveaux composants (appelés *Custom Components*). Ces derniers doivent aussi être conformes à la spécification générale d'un composant.

Le tableau 1.2 résume la description des 12 composants contenus dans un fichier CAP tel que présenté dans [Ora15b].

Numéro	Composant	Contenu
1	Header	Regroupe les informations générales sur le fichier CAP et le paquetage qui est défini
2	Directory	Liste la taille de chaque composant défini dans le fichier CAP. Contient aussi des entrées vers les nouveaux composants ajoutés (*custom components*)
3	Applet (optionnel)	Contient une entrée pour chaque applet contenue dans le fichier CAP. Si aucune applet n'est définie dans le paquetage, ce composant ne sera pas présent dans le fichier CAP.
4	Import	Liste l'ensemble des paquetages importés par les classes du paquetage défini (ce dernier ne figure pas dans la liste)
5	ConstantPool	Contient une entrée pour chaque : classe, méthode et champ référencé par les éléments du composant *Method* dans le fichier CAP.
6	Class	Décrit chacune des classes et interfaces définies dans le paquetage. Les classes représentées dans ce composant référencient d'autres entrées dans le même composant sous forme de : Superclass, Superinterface, références des interfaces implementées. Les classes représentées dans ce composant contiennent aussi des références vers des méthodes virtuelles définies dans le composant *Method* du fichier CAP.

7	Method	Décrit chaque méthode déclarée dans le paquetage, les méthodes abstraites définies par les classes, les handlers d'exception associés à chaque méthode.
8	StaticField	Contient toutes les informations requises pour créer et initialiser une image de tous les champs statiques définis dans le paquetage.
9	ReferenceLocation	Représente la liste des décalages dans le composant *Method* vers des éléments qui ont des points d'entrée dans le composant *Constant Pool*.
10	Export (optionnel)	Liste tous les éléments statiques qui peuvent être importés par des classes dans d'autres paquetages.
11	Descriptor	Donne les informations nécessaires pour analyser et vérifier tous les éléments du fichier CAP.
12	Debug (optionnel)	Contient toutes les méta-données nécessaires pour débuguer un paquetage sur une JCVM appropriée (ce composant n'est pas chargé dans la carte).

Tab 1.2 – Description des composants du fichier CAP (Extrait de [Ham12])

Les composants d'un fichier CAP se référencient entre eux (i.e. il y a de l'information partagée et redondante) [Ham12]. La figure 1.10 résume l'ensemble des liens d'interdépendance entre les différents composants.

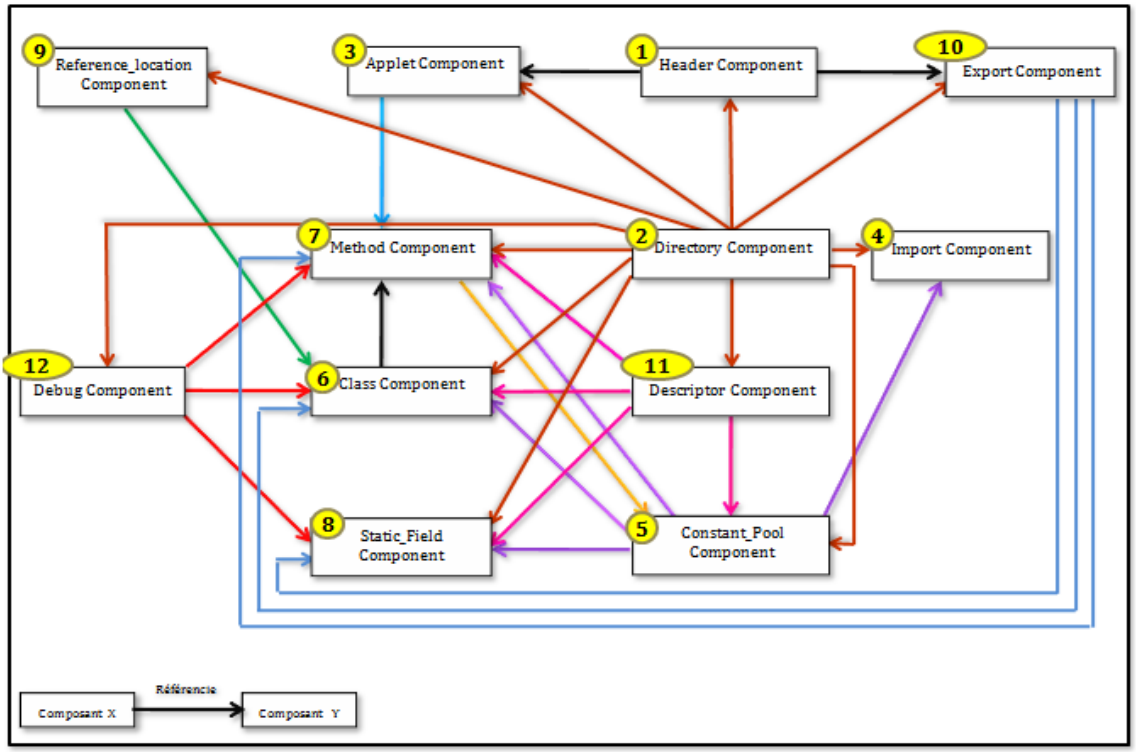

Fig 1.10 – Liens d'interdépendance entre les composants du fichier CAP (Extraite de [Ham12])

1.2.3.2 L'environnement d'exécution Java Card (JCRE)

Appelé aussi JCRE (pour Java Card Runtime Environment), il assure le rôle du système d'exploitation en gérant les ressources de la carte, la communication avec le réseau, l'exécution des applets et leur sécurité. Le JCRE sépare les applets de la propriété de la technologie des cartes à puce des vendeurs. Les applets sont ainsi plus faciles à écrire et sont indépendantes de l'architecture interne des cartes à puce.

1.2.3.3 APIs Java Card

Elles consistent en un ensemble de classes optimisées pour la programmation d'applications pour cartes à puce conformément à la norme ISO 7816. Les classes dans ces APIs incluent des classes adaptées à la plateforme Java Card fournissant un support pour le langage, des services de cryptographie ainsi que des classes spécialement conçues pour supporter la norme ISO 7816. Plusieurs classes Java ne sont pas nécessaires et donc non disponibles dans la plateforme Java Card. Une liste complète et détaillée des APIs est disponibles dans [Ora15c]. Entre autres, nous citons comme exemple les paquetages :

- `java.lang` : fournit les fondements pour le support du langage Java. Il supporte les classes `Object, Throwable` et `Exception` ;
- `javacard.framework` : apporte les classes et les interfaces essentielles pour programmer des applet Java Card. Il définit les classes : `Applet, APDU, JCSystem, OwnerPIN, AID, Util` ;
- `javacard.security` : fournit une architecture aux fonctions cryptographiques supportées par la plateforme Java Card ;
- `javacardx.crypto` : est un paquetage d'extension. Il définit une classe `Cipher` qui supporte les fonctions de chiffrement et de déchiffrement pour les différents algorithmes implémentés sur la carte ;
- `java.rmi` : définit l'interface `Remote` qui identifie les interfaces dont les méthodes peuvent être appelées par des applications clientes ;

1.2.4 Sécurité de Java Card

La sécurité offerte par la plateforme Java Card est la combinaison de la sécurité héritée du langage Java et des mécanismes intégrés à la plateforme elle même.

1.2.4.1 Sécurité du langage Java

Comme Java Card supporte un sous-ensemble du langage Java, approprié au développement des applications pour cartes à puce, elle en hérite les mécanismes de sécurité qui sont offerts, à savoir :

- Java est un langage fortement typé ce qui permet d'éviter les conversions de types non autorisées ;
- L'opération de cast suit des règles strictes. Un cast implicite d'un sous-type vers un super-type et un cast explicite obligatoire d'un type vers un sous-type.
- Java n'utilise pas d'arithmétique sur les pointeurs, il n'y a pas un moyen de forger des pointeurs ;
- Le niveau d'accès de toutes les classes, méthodes et champs est strictement contrôlé (`Public`, `Private`, `Protected`)

1.2.4.2 Sécurité de la plateforme

Elle est assurée par un ensemble de mécanismes fonctionnant de façon complémentaire en vue d'offrir une sécurité maximale à la carte. Les principaux mécanismes sécuritaires intégrés aux cartes Java Card 3.0 *Classic Edition* et versions antérieures sont [Ham12] :

Le vérifieur de byte code (BCV)

Ce composant (en anglais BCV pour ByteCode Verifier) intervient avant le chargement des applets sur la carte (processus offensif de la sécurité de Java Card). Il a pour objectif de s'assurer que le code à charger puisse être exécuté sans risque par la machine virtuelle et ne peut pas outrepasser les mécanismes de sécurité de haut niveau. La vérification consiste à effectuer une analyse statique (syntaxique et sémantique) du code à charger (i.e. le fichier CAP). Cette analyse assure par exemple : que ce fichier est bien formé, qu'il n'y a pas de débordement de pile, que le flot d'exécution reste confiné sur du bytecode valide, que chaque argument d'une instruction soit d'un type correct et que les appels de méthodes sont effectués conformément à leurs attributs de visibilité (`public, protected, private`) [Lan06]. Vu les ressources limitées des cartes à puce le processus de vérification de bytecode, coûteux en termes de temps d'exécution et d'espace mémoire, se fait dans la partie hors carte pour la plupart des cartes disponibles sur le marché. Ce qui ouvre une brèche de sécurité exploitable par les attaques contre les cartes à puce (section 1.3).

Le pare-feu

Il garantit l'isolation des applets et les différents objets créés au sein d'espaces protégés appelés *contextes* (figure 1.11). Un contexte est associé avec un paquetage de telle sorte que toutes les applets d'un même paquetage appartiennent au même contexte. De plus, le JCRE possède son propre contexte avec des privilèges spéciaux (les applets appartenant à ce contexte peuvent accéder aux objets de n'importe quel autre contexte de la carte). Outre le contrôle des droits d'accès, le pare-feu gère les changements de contextes. Á tout moment de l'exécution, il n y a qu'un seul contexte actif, il est appelé le *contexte courant*.

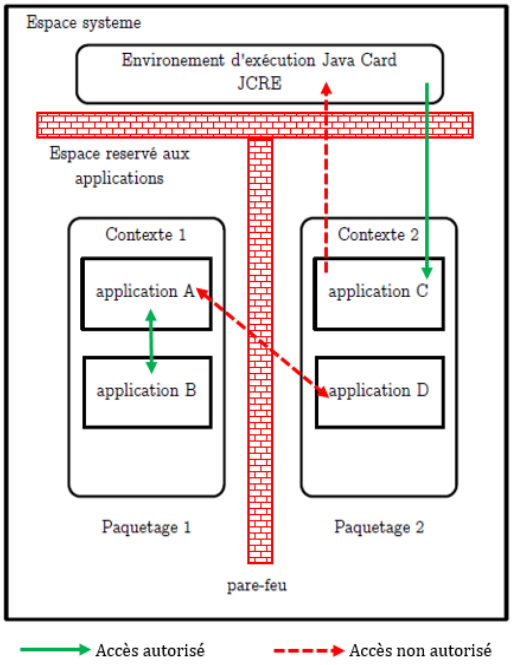

Fig 1.11 – Le pare-feu Java Card (adaptée de [Bar12])

Ainsi, sur la plateforme Java Card chaque objet créé appartient soit au contexte d'une applet soit à celui du JCRE, le propriétaire de l'objet créé étant l'applet du contexte courant. De plus, un objet est accessible seulement depuis le contexte de son propriétaire ce qui évite tout accès non autorisé à cet objet. Dans le cas où les applets ont besoin de partager des données ou d'accéder à des services du JCRE, des

mécanismes sécurisés de partage accessibles via les APIs spéciales (`javacard.framework.shareable`) sont utilisés.

Le mécanisme de transaction

Il permet de garantir l'intégrité des données lors de la mise à jour des objets dans la mémoire persistante de la carte. La mise à jour des champs des objets persistants doit âtre atomique (i.e totalement exécutée ou pas du tout) afin de palier à tout incident pouvant remettre en cause l'intégrité des données modifiées (par exemple une perte d'alimentation). L'atomicité est garantie dans Java Card à l'aide d'un mécanisme de transaction qui s'assure que *toutes ou aucune* des opérations à l'intérieur d'un bloc soient terminées. Le déclenchement, la terminaison ou l'annulation des transactions se fait via des méthodes des APIs (respectivement `beginTransaction(), commitTransaction()` et `abortTransaction()` de la classe `JCSystem`). Le mécanisme de `rollback` des données est utilisé pour restaurer les anciennes données en cas d'échec ou en cas d'annulation de la transaction.

1.2.4.3 GlobalPlatform

L'utilisation des cartes multi-applicatives (dont Java Card en fait partie) à grande échelle est régie par les spécifications GlobalPlatform [9]. Elles fournissent une architecture globale de gestion de la sécurité et des cartes. Le but de ces spécifications est d'apporter un standard pour gérer les cartes de façon indépendante du matériel, des vendeurs et des applications.
L'architecture GlobalPlatform comprend un certain nombre de composants permettant de s'abstraire du matériel du vendeur grâce à des interfaces pour les applications (APIs standardisées) et pour le système de gestion hors carte (APDUs standardisées). Principalement, nous trouvons les composants suivants :

1. *Le gestionnaire de la carte (Card Manager)* : Il joue le rôle d'administrateur central de la carte, c'est le représentant de l'émetteur de la carte. Il a la possibilité de charger, d'installer et d'effacer les applications appartenant à l'émetteur ou à d'autres fournisseurs d'applications.
2. *Les domaines de sécurité (Security Domains)* : Sont les représentants des fournisseurs d'applications sur la carte. Ils proposent des services de sécurité comme la manipulation des clés, le chiffrement, le déchiffrement, la génération et la vérification des signatures. Chaque domaine de sécurité implémente un protocole de canal sécurisé (SCP pour Secure Channel Protocol) qui permet de sécuriser la communication entre l'émetteur de carte, le fournisseur d'applications, ou l'autorité de contrôle et son domaine de sécurité sur la carte. Chacun de ces acteurs possède ses propres clés et qui sont complètement isolées de celles des autres domaines de sécurité.
3. *Les APIs de GlobalPlatform* : Elles fournissent aux applications des services de vérification du détenteur de la carte, personnalisation, services sécuritaires ainsi que des services de gestion du contenu de la carte (blocage de la carte ou mise à jour des états du cycle de vie de l'application).

1.3 Les attaques contre les cartes à puce

Une attaque consiste à utiliser les caractéristiques ou les particularités de la cible à attaquer afin de tenter de contourner les mécanismes de sécurité matériels ou logiciels qui lui sont intégrés [Sér10]. Le but d'un attaquant est de pouvoir accéder aux informations et aux secrets contenus dans la carte (code PIN, clés secrètes, etc) ou tout simplement de nuire aux applications embarquées afin de tester le niveau sécuritaire de la carte [NSIcL09].
La sécurité d'une carte à puce peut être contournée de plusieurs manières : soit en prenant le matériel

9. `https://globalplatform.org/specs-library/`

en défaut (section 1.3.1), soit en prenant l'applicatif ou le système en défaut (section 1.3.2), ou bien en combinant les deux (section 1.3.3). Ainsi nous pouvons classer les attaques connues contre cartes à puce selon ces trois catégories (figure 1.12). Dans les sections suivantes, nous présenterons brièvement chacune de ces classes d'attaques.

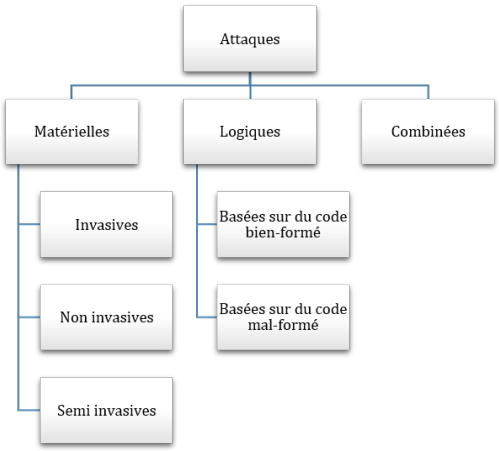

Fig 1.12 – Classification des attaques contre cartes à puce

1.3.1 Attaques physiques ou matérielles

Ces attaques touchent la partie matérielle de la carte à puce. Elles consistent en une analyse et/ou modification des circuits électroniques de la carte afin d'obtenir des informations sensibles de cette dernière. Ces attaques peuvent être : *invasives*, *non invasives* ou *semi invasives*.

1.3.1.1 Attaques invasives

Ces attaques consistent à un accès physique aux composants internes de la carte afin de réaliser des observations directes ou encore des modifications de la structure des circuits. Il s'agit d'attaques extrêmement performantes, qui présentent cependant l'inconvénient majeur d'être destructive : une fois l'attaque effectuée, la carte devient inutilisable. Le but est de récupérer un ensemble d'informations de la carte en se basant sur une cartographie des circuits. Pour arriver à cela, il faut une étape d'isolation physique ou chimique des circuits de la carte (d'où l'aspect *invasif*). Plusieurs méthodes pour réaliser de telles attaques sont [Wit02, Ham12] :

— *Produits chimiques et gravure à eau forte (Etching).* Les produits chimiques sont utilisés en premier pour dissoudre la couche de résine qui fixe les circuits. Par la suite, la technique du *Etching* permet de décapsuler les circuits et isoler les différentes couches dans le but d'obtenir tous les blocs fonctionnels d'un circuit afin qu'il soit accessible pour analyse.

— *Scanning Electron Microscopes (SEM).* Cette technique permet de reconstruire des circuits complets ou le code source du système d'exploitation à partir du contenu des cellules de la ROM en effectuant l'analyse optique et le reverse engineering. De plus, elle permet d'observer des circuits durant une exécution dans le cas où ce dernier a été soigneusement préparé et donc peut toujours effectuer ses fonctions. Ce qui permet de retrouver des sections de code qui sont actives ainsi que les valeurs des cellules de la mémoire.

— *Le Probing.* Cette technique permet de positionner des micro-sondes, arbitrairement sur les fils d'un circuit isolé mais qui réalise encore ses fonctions. Ce qui permet de créer de nouveaux canaux vers l'extérieur de la carte. Les conséquences de l'application de cette technique dépendent de

la zone où les sondes sont posées. Si par exemple, la cible est le bus de données, les sondes permettront de capter tous les échanges de données entre le CPU et les mémoires. Avec une analyse poussée, ceci permet d'obtenir le code du programme en exécution ainsi que les clés incluses dans les programmes ou encore détourner l'exécution du CPU.

— *FIB (Focused Ion Beam)*. Cette technique permet de modifier les circuits en rajoutant ou éliminant des pistes conductrices à l'aide de rayons d'ions. Ceci a pour conséquence : la reconnexion de circuits séparés, l'envoi de signaux internes cachés vers l'extérieur, l'élargissement des pistes fines et fragiles pour déposer des sondes, *etc.*

Les attaques invasives sont les plus coûteuses car elles doivent être menées par des experts qui manipulent des équipements sophistiqués et coûteux.

1.3.1.2 Attaques non invasives

Ces attaques ne nécessitent pas de détruire la carte à puce cible. Elles sont basées sur l'observation de canaux auxiliaires, ou canaux cachés, d'où le nom *attaques par canaux cachés*. Ces attaques visent à analyser une grande quantité de données observées (à partir des traces émises par le composant) afin de filtrer les informations essentielles (renseignement de façon indirecte sur son comportement). Les attaques par canaux cachés ont donné lieu à une littérature très riche depuis la fin des années 90 dans le domaine de la carte à puce, en particulier grâce aux travaux de Kocher et al. [Koc96, KJJ+98, KJJ99]. Les phénomènes physiques utilisés pour observer le comportement d'un circuit électronique peuvent être : la consommation du courant, les radiations électromagnétiques, le temps d'exécution du CPU.

— **Attaques par analyse de courant**. Elles sont basées sur l'observation de la consommation d'énergie d'un circuit dans le but de retrouver de l'information sur les opérations effectuées ainsi que sur les valeurs des données manipulées par ces opérations. En effet, la consommation électrique d'un module embarqué, à un instant précis, dépend de : l'instruction exécutée, des opérandes (adresses ou valeurs) manipulés ainsi que de l'état précédent du module. Les attaques par analyse de courant sont divisées en deux catégories :
  — Les attaques par analyse élémentaire de consommation appelées aussi **SPA** (*Simple Power Analysis*) [MDS99, Man02, Osw02, CMW14]. Elles consistent à mesurer directement la consommation du courant d'un composant sécurisé avec un oscilloscope durant l'exécution du code, afin d'obtenir des informations sur les instructions exécutées ou sur les données traitées (qui peuvent être des clés cryptographiques par exemple).
  — Les attaques par analyse différentielle de consommation appelées aussi **DPA** (*Differential Power Analysis*) [JJK99, DBLW02]. Elles permettent d'obtenir de l'information en utilisant des méthodes statistiques permettant de distinguer la faible corrélation qui existe entre la valeur des données manipulées et les mesures de consommation électrique relevées.
— **Attaques par analyse du rayonnement électromagnétique**. Appelées aussi **EMA** (*ElectroMagnetic Analysis*) [SQ01, MOG01]. Ces attaques sont très similaires à celles par analyse de courant. En effet, le courant utilisé par la puce crée un champ électromagnétique qui dépend lui aussi de l'activité de la puce. Ce champ peut être mesuré grâce à une sonde spécifique reliée à un oscilloscope et l'analyse de ce signal s'effectue ensuite de manière similaire à l'analyse de la consommation électrique. Cependant, une telle analyse permet d'isoler l'activité d'une toute petite partie de la puce. En effet, suivant la position de la sonde à la surface du composant, l'attaquant va pouvoir analyser le rayonnement électromagnétique de telle ou telle partie du composant. Par conséquent, le bruit induit par les autres parties de la puce lors d'une analyse de consommation électrique peut être grandement diminué.

— **Attaques par analyse du temps d'exécution** [Koc96, Sch02]. Elles sont basées sur le temps d'exécution des instructions dans le but de déduire des informations secrètes telles que les clés cryptographiques manipulées par un programme donné. En effet, de telles attaques reposent sur la constatation que le temps d'exécution d'un algorithme est souvent dépendant de ses données, et que l'observation du temps d'exécution permet donc de déterminer un certain nombre d'informations sur les données, y compris quand il s'agit de données protégées en confidentialité par la carte.

1.3.1.3 Attaques semi invasives

Ces attaques modifient le matériel sans le détruire. Elles sont appelées *attaques par perturbation* ou *attaques par injection de fautes*. Elles consistent à modifier un ou plusieurs paramètres physiques de l'environnement afin de perturber le fonctionnement normal du composant sécurisé. Ceci dans le but d'exécuter des opérations normalement sécurisées (i.e. que normalement l'attaquant n'est pas autorisé à les effectuer) ou d'accéder à des données sensibles et secrètes de la carte à puce.
Comme les attaques par injection de faute constituent le moyen d'activation (le déclencheur) des codes malveillants faisant l'objet de notre travail, elles seront présentées en détails dans le chapitre 3 qui est y dédié.

1.3.2 Attaques logiques

Avec l'apparition des cartes ouvertes, permettant de charger des applications dans la carte après son émission, les *attaques logiques* sont de plus en plus répandues. Elles utilisent des failles pour contourner les protections mises en place [NSIcL09]. Ceci dans le but d'obtenir des données confidentielles ou effectuer des modifications non autorisées aux données de la carte. Les attaques logiques ne sont pas destructives (i.e. la partie matérielle de la carte n'est pas affectée) et sont facilement reproductibles. Elles sont moins coûteuses que les attaques physiques car elles nécessitent un minimum d'équipement : une carte, un lecteur et un ordinateur. Par contre, leur succès nécessite une grande compétence, persistance et imagination des attaquants afin de trouver des vulnérabilités et les exploiter [Ham12].
Comme une Java Card peut héberger des applets issues de divers fournisseurs, il devient possible d'avoir deux applets en provenance de deux fournisseurs concurrents qui co-existent dans la même carte. D'après Witteman [Wit03], les applets peuvent avoir certains comportements indésirables (classés par ordre de gravité) :

— Comportement inoffensif mais embarrassant ;
— Crash de la carte (de façon temporaire ou permanente) ;
— Déclenchement d'un comportement externe non autorisé (à l'aide de moyens externes) ;
— Révélation de la confidentialité de l'utilisateur ;
— Attaque d'autres applets de la carte ou la plateforme elle même.

C'est la dernière catégorie (*applet malicieuse*) qui présente le plus de risques pour Java Card. Une applet malicieuse peut être de deux types [Raz16] :

— **Applet malicieuse bien-formée** : Elle consiste en une application ayant un comportement malveillant, mais reste structurellement et sémantiquement correcte vis-à-vis de la spécification Java Card (d'où le nom bien-formée). Elle a la capacité d'effectuer explicitement des opérations malveillantes telles que, par exemple, exploiter des bugs de la plateforme ou simplement en renvoyant des données sensibles vers l'extérieur.
— **Applet malicieuse mal-formée** : Il s'agit d'une applet (ou plus exactement le fichier CAP correspondant) qui a été modifiée afin d'intégrer une séquence particulière de bytecode qui

n'est généralement pas produite par un compilateur. La caractéristique principale d'une applet mal-formée est qu'après sa modification, elle ne sera plus structurellement ou sémantiquement correcte. Par conséquent, il est peu probable qu'un tel code passe les vérifications effectuées par un vérifieur de bytecode embarqué ou non.

Afin de classer les attaques logiques, nous nous sommes basés sur les classifications proposées par [Raz16, FL16] (une fusion des deux). En résumé, une attaque logique peut être basée sur :

- Un code [10] bien-formé. Les applications peuvent être vérifiées (leur code est correct), et peuvent être chargées sur des cartes réelles. Cependant, ces attaques sont spécifiques à une plateforme donnée. Une telle attaque exploite :
  - Les points faibles dans la spécification Java Card (ambiguité de la spécification ou mauvaise compréhension des développeurs)
  - Une mauvaise implémentation de la plateforme Java Card .
- Un code mal-formé qui compromet la plateforme en arrivant à se charger dans la carte sans qu'il ne soit vérifié et procède à attaquer d'autres applets ou la plateforme elle même. Ces attaques sont assez puissantes et elles donnent de bons résultats sur de nombreuses plateformes. Elles ont comme hypothèse que le code peut être chargé sans vérification (i.e. l'attaquant dispose des clés de chargement + absence du BCV).

Catégories d'attaques logiques		**Exemples d'attaques logiques**
Code bien-formé	Mauvaise compréhension de la spécification	— Abus du mécanisme de transaction [MP08] — Abus du pare-feu [Wit03, MP07, MP08]
	Mauvaise implémentation de la plateforme	— Attaques contre le vérifieur de bytecode [LB15, LB16] — Dépassement de pile et changement du contexte de sécurité [Dub16]

Code mal-formé		— Attaques par confusion de type [Wit03, Hyp03, MP08, ICL09, BTG10, BICL11]
		— EMAN2 : changement de l'adresse de retour [BICL11]
		— Dépassement de pile :
		  — Abus de l'instruction `dup_x` [Fau13]
		  — Abus du mécanisme de création de la frame [LR15]
		— Changement du flot de contrôle (`jsr/ret`)[BL15]
		— Attaque de l'API `ArrayCopyNonAtomic` [FL17]
		— Attaques contre l'édition statique des liens [Hyp03, ICL09, HBL+12]

Tab 1.3 – Catégories et exemples d'attaques logiques

Comme les attaques logiques ne font pas l'objet de notre travail, on se limite à donner une vue synthétisée en citant quelques exemples (le tableau 1.3) d'attaques logiques existantes, qui appartiennent

10. Nous utiliserons indifféremment les termes code et applet

aux catégories définies ci-dessus, ainsi que les références correspondantes pour plus de détails techniques sur leur mise en œuvre. En outre, le lecteur peut se référer aux travaux [Bar12, Bou14, Raz16, FL16] pour une synthèse sur ces attaques, les contremesures associées ainsi que d'autres exemples.

1.3.3 Attaques combinées

C'est la nouvelle tendance des attaques contre cartes à puce. Comme leur nom l'indique, ces attaques combinent les attaques physiques et logiques et donc profitent des avantages de ces deux types d'attaques. Le but étant de contourner les mécanismes de sécurité embarqués qui ont tant rendu les précédentes attaques logiques difficiles à réaliser avec succès. Ainsi il est devenu possible d'exploiter un ensemble plus large des vulnérabilités identifiées, chose qui alourdit l'impact sur la sécurité des cartes à puce [Ham12].
L'idée est de charger une application malveillante qui peut passer les mécanismes de vérification sans qu'elle ne soit détectée. Une fois sur la carte, et suite à une injection de faute (attaque physique) qui fait muter le code de l'application, cette dernière change de comportement (pour réaliser une attaque logique). Des exemples d'attaques combinées contre les cartes à puce Java Card sont présentés dans la section 3.6.

1.4 Conclusion

Le but de ce chapitre était de survoler les aspects de base liés au contexte général de notre travail. L'idée est de partir du général au particulier, d'où la décomposition en trois principaux points. Au départ, nous avons présenté la carte à puce en tant que *support d'exécution sécurisé* pour notre travail : La typologie des cartes, les systèmes d'exploitation pour cartes ainsi que le mode de communication avec le monde extérieur ont été passés en revue. Par la suite, nous avons abordé les concepts liés à notre *plateforme cible*, à savoir Java Card. En effet, cette technologie vise à offrir un environnement de développement simple et très sécurisé dans un dispositif à ressources limitées comme la carte à puce d'où son intégration dans la plupart des cartes dans le monde. Cependant, vu la nature des données sensibles que les cartes détiennent, elles sont devenues la cible de plusieurs types d'attaques. L'accent a été mis sur cet aspect dans la dernière partie de ce chapitre. Ceci, en présentant un panorama, non exhaustif, des différentes attaques auxquelles une carte à puce pourrait faire face.

Ceci fait, l'objectif des deux chapitres suivants est de présenter un état de l'art des travaux existants concernant deux questions fondamentales relatives à notre problématique :

— *Q1 : Comment un code (indépendamment de sa nature hostile ou non) pourrait être dissimulé ?* C'est le but du chapitre 2 dans lequel nous passerons en revue les différentes techniques d'obfuscation de code en accordant une importance aux techniques liées aux codes malveillants.
— *Q2 : Comment activer un code caché ?* C'est dans le chapitre 3 que nous allons reprendre les attaques par injection de faute en tant que déclencheurs des attaques combinées (catégorie d'attaque à laquelle appartient notre travail).

CHAPITRE 2

Obfuscation de code

Sommaire

L'obfuscation consiste à appliquer des transformations sur un programme afin d'obtenir de nouvelles versions plus difficiles à comprendre et à analyser manuellement ou par des outils automatiques, tout en préservant sa sémantique. Comme l'obfuscation a été étudiée pendant plus de deux décennies, une multitude de travaux traitant différents aspect de ce concept sont disponibles.

L'étude novatrice a été présentée en 1997 par Collberg et al. [CTL97] qui ont proposé une taxonomie détaillée de plusieurs techniques d'obfuscation. Cette étude est devenue la base de la majorité des recherches qui ont suivi. De nombreux travaux traitent ce sujet mais avec des objectifs différents.

Drape et al. [Dra09] ont présenté plusieurs transformations d'ordre général et d'autres qui sont dépendantes du language traité.

Balakrishnan et Schulze [BS05] ont étudié plusieurs techniques d'obfuscation applicables pour les codes bénins et les codes malveillants.

Xu et al. [XZKL17] ont étudié les approches existantes pour l'obfuscation orientée code et l'obfuscation

orientée modèle avec une étude comparative des deux classes.
Faruki et al. [FFL+16] ont passé en revue les différentes techniques d'obfuscation de code ainsi que les outils existants tout en mettant l'accent sur les applications Android. En outre, ils ont analysé les techniques utilisées par les auteurs de programmes malveillants en vue d'échapper aux processus d'analyse.
Hosseinzadeh et al. [HRL+18] ont fait une étude assez conséquente sur l'obfuscation et la diversification de programmes comme deux techniques d'amélioration de la sécurité des logiciels. Ils ont adopté une méthode de revue systématique de la littérature (en anglais SLR pour *systematic literature review*). A la suite de cette recherche systématique, ils ont collecté 357 articles en rapport avec le sujet d'intérêt, publiés entre 1993 et 2017. Ils ont étudié les articles collectés, analysé les données qui y sont extraites et présenté une classification de ces dernières. Ceci, dans le but de donner un état de l'art cohérent et assez complet des deux techniques, éclairer les lacunes de la recherche et fournir une base pour les futurs travaux de recherche dans le domaine.
L'obfuscation est utilisée dans divers domaines et plusieurs transformations (générales ou spécifiques) sont applicables aux différents niveaux d'un programme. Dans ce qui suit, nous présentons un aperçu de ces différents aspects liés à l'obfuscation (d'ordres général et spécial pour les malware).

2.1 Définition du concept

La définition la plus générale de l'obfuscation de code est présentée dans les travaux de Collberg et al. [CTL97]. Ils ont formalisé la notion de transformation d'obfuscation comme suit :

Définition 2.1. *(transformation d'obfuscation [CTL97]) Soit $\tau : P \rightarrow P'$ une fonction transformant un programme P en un programme P'. τ est une transformation d'obfuscation de code si $\tau(P)$ possède le même comportement observable que P, sachant que :*

- *si le programme P ne se termine pas ou s'il se termine avec une erreur, alors le programme $\tau(P)$ peut éventuellement se terminer ou non ;*
- *dans le cas contraire (cas où le programme P se termine), le programme $\tau(P)$ se termine aussi en fournissant la même sortie que P.*

2.2 Applications

L'obfuscation du code est largement utilisée dans la pratique. Les techniques existantes sont en général utilisées pour l'un ou plusieurs des objectifs suivants [Eyr17] :

- Protéger la propriété intellectuelle des concurrents en rendant le reverse-engineering très difficile.
- Protéger la gestion des droits numériques (GDN) (en anglais DRM pour Digital Rights Management) des ressources multimédias afin de réduire le piratage.
- Les auteurs de malware utilisent l'obfuscation pour cacher leurs créations des anti-malware et des analyses approfondies le plus longtemps possible, de manière à ce qu'ils puissent se propager et infecter de plus en plus de périphériques.
- Empêcher ou au moins retarder les analystes humains ou les moteurs d'analyse automatiques de déterminer l'intention du code malveillant.

2.3 Les critères d'évaluation de la qualité de l'obfuscation

Collberg et al. [CTL97] et Low [Low98] proposent d'évaluer la qualité des transformations d'obfuscation selon quatre critères :

— La *puissance* mesure le niveau de difficulté de compréhension du code obfusqué par rapport au code original pour un analyste humain.
— La *résilience* mesure la résistance d'une transformation face à un déobfuscateur automatique.
— La *furtivité* détermine à quel point le code obfusqué se fond avec le reste le programme (i.e. difficilement dissociable).
— le *coût d'exécution* concerne la consommation supplémentaire en termes de temps CPU/espace mémoire qui est ajoutée au programme d'origine ;

Par conséquent, la qualité d'une transformation d'obfuscation est définie comme étant une combinaison des valeurs des métriques précédentes. Le but étant de trouver un compromis entre un coût réduit et une valeur élevée pour les autres critères. Certaines mesures, issues du domaine de l'ingénierie logicielle, conçues pour quantifier la qualité d'un programme sont utilisées par Collberg et al. [CTL97] pour évaluer la puissance d'une transformation, comme par exemple : la longueur du programme, la complexité cyclomatique, la complexité du flot de données ou encore celle de la structure des données. Une présentation plus complète et détaillée des mesures de complexité qui permettent de définir la puissance d'une transformation d'obfuscation est proposée par Cornelie dans [Cor16].

2.4 Les niveaux d'obfuscation

Les transformations d'obfuscation peuvent s'appliquer aux différentes représentations d'un programme, principalement le code source, la représentation intermédiaire (IR) ou le niveau assembleur. De plus, des combinaisons des différentes transformations sont possibles vu que ces dernières peuvent être appliquées séquentiellement sur un même fragment de code au cours de son processus de compilation [Cap12].

2.4.1 Niveau de code source

Une obfuscation à ce niveau exploite les spécificités du langage de programmation en entrée. Comme l'étape d'obfuscation a lieu avant la phase de compilation, il est plus facile de l'intégrer dans une chaîne de compilation existante. Dans Madou et al. [MADB+06], les auteurs effectuent des transformations de haut niveau, compilent l'application cible et observent l'effet de l'obfuscation au niveau binaire. Soutenus par des résultats empiriques, ils concluent que plusieurs transformations survivent aux modifications établies par le compilateur. Par conséquent, il n'est pas toujours nécessaire d'appliquer des étapes supplémentaires d'obfuscation au niveau binaire sur des applications qui ont déjà subi des transformations au niveau code source. Collberg et al. ont largement décrit les techniques disponibles pour obfuscation du code source dans [CTL97, CTL98]. Des exemples de tels obfuscateurs sont :

— Pour Java/Android : DashO [1], DexProtector [2], ProGuard [3], DexGuard [4], Allatori [5],
— Pour C/C++ : Obfuscateur C de Semantics Designs [6], Obfuscateur C++ de StarForce [7], CXX-Obfus de Stunnix [8]
— Pour .NET : Dotfuscator [9]

1. https ://www.preemptive.com/products/dasho/overview
2. http ://dexprotector.com
3. https ://www.guardsquare.com/proguard
4. https ://www.guardsquare.com/dexguard
5. http ://www.allatori.com
6. http ://www.semanticdesigns.com/Products/Obfuscators/CObfuscator.html
7. http ://www.star-force.com/products/starforce-obfuscator/
8. http ://stunnix.com/prod/cxxo/
9. http ://www.preemptive.com/products/dotfuscator/

2.4.2 Niveau de représentation intermédiaire

La représentation intermédiaire (IR) est conçue pour être indépendante de tout langage source ou cible. Ainsi, les obfuscateurs opérant à ce niveau sont plus généraux que les obfuscateurs au niveau source. Ils peuvent traiter des programmes issus de différents langages source. Cependant, l'intégration des transformations est plus difficile car il faut ajouter l'obfuscateur à la chaîne de compilation existante [Eyr17]. Obfuscator-LLVM [10] est un exemple d'obfuscateur fonctionnant sur le code LLVM Intermediate Representation (IR) [JRWM15]. nous citons aussi Epona [11] qui est un obfuscateur commercial développé par Quarkslab.

2.4.3 Niveau assembleur

Le niveau d'assembleur présente une perte d'information majeure par rapport aux niveaux IR et source, il est donc très difficile de mettre en œuvre un obfuscateur général ne fonctionnant que sur le langage assembleur [Eyr17]. Une technique possible consiste à appliquer une protection par virtualisation directement sur des programmes binaires. Le code protégé s'exécute ensuite sur un processeur virtuel différent des processeurs standards. VMProtect [12] est un obfuscateur commercial implémentant la virtualisation. Une autre technique associée à l'obfuscation à bas niveau est la randomisation de l'ensemble d'instructions (ISR pour Instruction Set Randomisation) [KKP03], [BAFS05]. Un environnement d'exécution unique est créé pour le processus en cours d'exécution dans le système. En d'autres termes, un nouveau jeu d'instructions est créé pour chaque processus. Par conséquent, l'attaquant ne connaît pas le langage utilisé et donc ne peut pas communiquer avec la machine.

2.5 Classification des techniques d'obfuscation

La classification des techniques d'obfuscation telle que présentée par Collberg et al. [CTL97] est basée sur la cible de la transformation. En d'autres termes, ce qui est transformé et comment la transformation est appliquée. Les quatre catégories définies pour les transformations d'obfuscation sont : la structure lexicale, les données, le flot de contrôle et les transformations préventives. Loin d'être exhaustive, cette section donne un bref aperçu sur quelques techniques d'obfuscation existantes (la figure 2.1 résume la classification qui sera suivie dans ce qui suit). L'accent est mis sur les trois premières catégories vu que ce sont les plus discutées dans la littérature. Pour plus de détails sur ces techniques et des exemples supplémentaires, le lecteur peut se référer aux travaux [CTL97, Dra09, FFL+16, HRL+18]

2.5.1 Transformations de la structure lexicale

Elles visent à changer l'apparence d'un programme tout en gardant sa sémantique intacte. En réduisant les informations disponibles pour un lecteur humain, le reverse-engineering d'un programme devient plus difficile [HRL+18]. Ces transformations consistent, par exemple à :

- Renommer tous les identificateurs (variables, classes, méthodes, données sensibles, etc) en choisissant des noms peu significatifs.
- Supprimer les commentaires et les informations sur le débogage.
- Changer le formatage du code source (les espaces inutiles, éliminer l'indentation, *etc*).

10. https ://github.com/obfuscator-llvm/obfuscator/
11. https ://epona.quarkslab.com/
12. http ://vmpsoft.com/

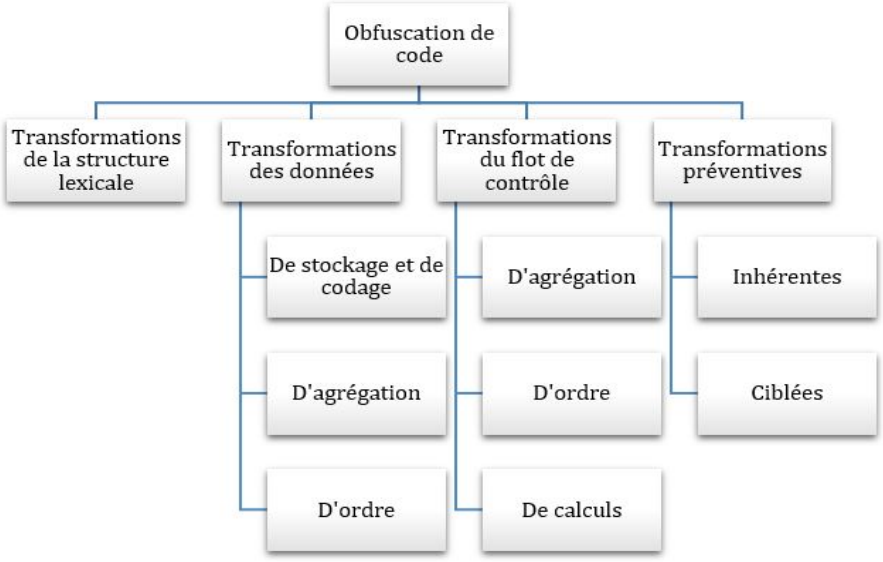

Fig 2.1 – Classification des transformations d'obfuscation (adaptée de [CTL97])

2.5.2 Transformations des données

Elles ont pour objectif de masquer les données et les structures de données qu'un programme peut utiliser. Les valeurs prises lors de l'exécution ainsi que les informations pouvant être déduites de l'organisation et des interactions des données sont masquées [Eyr17]. Diverses approches ont été proposées pour atteindre cet objectif. Selon Collberg et al. [CTL97], ces transformations concernent : le *stockage et le codage* qui modifient la représentation des données, *l'agrégation* qui scinde / fusionne des données et *l'ordre* qui permet de permuter les éléments dans les structures de données existantes. Des exemples de transformations de données sont :

- Changer une variable simple par une expression à évaluer (des exemples de transformations possibles sont présentés en détails dans [Dra04]).
- Fusionner plusieurs variables scalaires en une seule variable.
- Promotion de variables, i.e. transformer des variables scalaires en objets plus complexes.
- Transformation des tableaux : découpage d'un tableau en plusieurs sous-tableaux, fusion de plusieurs tableaux en un seul tableau, aplatissement (à l'inverse augmentation) des dimensions d'un tableau, réorganisation des éléments d'un tableau (en appliquant une fonction de permutation).

2.5.3 Transformations du flot de contrôle

Elles visent à accroître l'obscurité du flot de contrôle des programmes. Il existe de nombreuses recherches portant sur les techniques d'obfuscation du flot de contrôle. Elles affectent principalement l'agrégation, l'ordre ou les calculs du flot de contrôle [CTL97]. Les transformations *d'agrégation* divisent les calculs qui sont liés logiquement et fusionnent ceux qui sont indépendants. Les transformations de *contrôle* réorganisent les blocs de code, les boucles et les expressions en préservant leurs dépendances. Les transformations de *calcul* insèrent un nouveau code ou apportent des modifications algorithmiques au code source. Avant de présenter des exemples des transformations du flot de contrôle, nous avons jugé nécessaire d'introduire deux notions principales, à savoir : le flot de contrôle et les prédicats opaques.

2.5.3.1 La notion de flot de contrôle

Le flot de contrôle d'un programme désigne tous ses chemins d'exécution possibles, et comment ces chemins sont choisis. Il peut être représenté avec deux graphiques :

— Le *graphe de flot de contrôle* (GFC, en anglais CFG pour *Control Flot Graph*) qui est spécifique à une seule fonction et représente toutes les possibilités exécutions de cette dernière. Chaque nœud du graphe est un bloc de base : une suite d'instructions continue, sans saut ni cible de saut. Un arc entre deux blocs de base signifie qu'il existe un chemin d'exécution possible reliant ces blocs.

— Le *graphe d'appel* qui donne des informations sur le déroulement du programme. Chaque nœud du graphe représente une fonction, un arc partant d'un nœud f1 à un nœud f2 signifie que la fonction f1 contient au moins un appel de la fonction f2.

2.5.3.2 La notion de prédicat opaque

Concept introduit par Collberg et al. dans [CTL97], il désigne un prédicat (une expression booléenne) dont la sortie est connue à l'étape d'obfuscation mais qui est difficile à déduire par la suite (voir définition 2.2). D'après Collberg et al. [CTL98], les prédicats opaques constituent la base de la conception d'une grande partie des transformations de contrôle. De ce fait, la qualité de ces dernières dépend directement de la qualité des prédicats employés.

Définition 2.2. *(prédicat opaque [CTL97]). Un prédicat est dit opaque si sa valeur est connue au moment de la transformation du programme mais qu'elle est difficile à déduire pour un outil de désobfuscation. Un prédicat vrai (faux), que nous noterons P^T (P^F) est toujours évalué à vrai (faux). Nous pouvons également définir un prédicat $P^?$ dont la valeur est délibérément inconnue ou n'influe pas sur l'exécution du programme.*

Un cas de figure classique utilisant des prédicats opaques consiste à créer de fausses branches dans le GFC d'une fonction. La valeur d'un prédicat opaque P^T étant toujours à vrai, la branche correspondant à son évaluation à faux ne sera jamais activée. Donc n'importe quel code peut être associé à cette deuxième branche du fait qu'il ne sera jamais exécuté (code mort). La figure 2.2 illustre cet exemple. La ligne en pointillée indique un chemin qui ne sera jamais pris, quelle que soit l'entrée du programme.

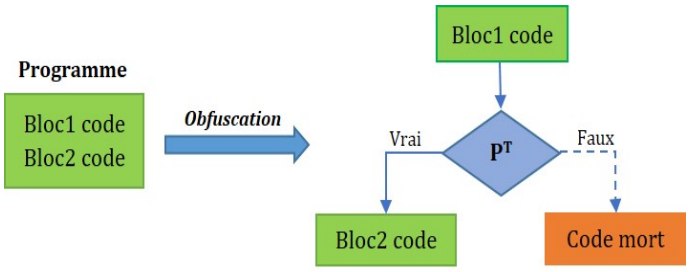

Fig 2.2 – Exemple classique de de prédicat opaque

Pour la construction des prédicats opaques, plusieurs techniques ont été proposées dans la littérature. Elles peuvent être basées sur des résultats de la théorie des nombres (la figure 2.3 présente quelques exemples d'expressions arithméthiques évaluées toujours à vrai), ou sur des informations difficiles à obtenir lors d'une phase d'analyse (par exemple : utilisation des alias de pointeurs et structures dynamiques, utilisation des threads). Plus d'exemples et de détails sur les prédicats opaques sont dans [CTL98, Dav17].

$$\forall x, y \in Z : 7y^2 - 1 \neq x^2$$
$$\forall x \in Z : 3|(x^3 - x)$$
$$\forall x \in N : 14|(3.7^{4x+2} + 5.4^{2x-1} - 5)$$
$$\forall x \in Z : 2|x \vee 8|(x^2 - 1)$$
$$\forall x \in N : 2| \left[\frac{x^2}{2}\right]$$

Fig 2.3 – Exemples de prédicats opaques toujours vrais (extraite de [Arb02])

2.5.3.3 Exemples de transformations du flot de contrôle

A. **Aplatissement du flot de contrôle**

L'idée de l'aplatissement du flot de contrôle consiste à changer complètement la structure d'un GFC de la fonction, en codant les informations du flot de contrôle dans le flot de données. Les blocs basiques se retrouvent au même niveau, le contrôle du flot se fait à l'aide d'un nœud spécial nommé « dispatcher ». Ce dernier gère l'exécution avec une variable (`switch_variable`) dont la valeur détermine quel bloc doit être exécuté après le bloc actuel. À la fin de chaque bloc de base, cette variable est mise à jour et le contrôle retourne au dispatcher qui décide de la suite.

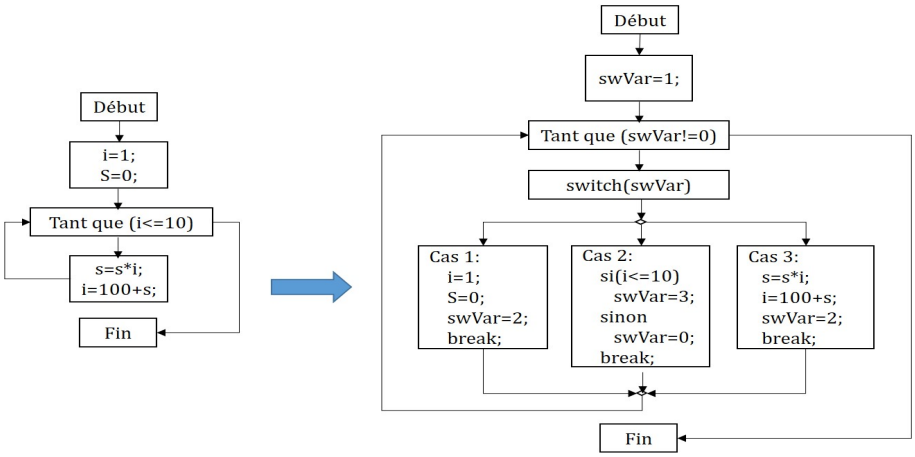

Fig 2.4 – Exemple d'aplatissement du flot de contrôle (Extraite de [Cor16])

B. **Transformations des boucles**

Plusieurs techniques peuvent être appliquées pour changer la structure des boucles dans le but d'augmenter la complexité du programme original. Le travail de Bacon [BGS94], portant sur les techniques employées par les compilateurs pour la reconstruction des programmes de haut niveau, a été la base pour les techniques proposées dans [CTL97]. Parmi ces dernières, nous citons (figure 2.5) :

— *Extension de la condition de la boucle* : consiste à masquer la condition de terminaison de la boucle. L'idée est d'utiliser des prédicats opaques qui ne vont pas changer le nombre d'itérations. Par exemple : une condition `cond1` peut être remplacée par une condition `cond2` qui vaut `(cond1 & P)` avec P un prédicat opaque qui est toujours à *vrai*.

— *Découpage de boucle* : consiste à découper l'espace d'itération de manière à ce que le corps de la boucle tienne dans le cache. Ceci améliore le comportement de la boucle vis- à-vis du cache.

— *Déroulage de boucle* : consiste à dupliquer le corps de la boucle une ou plusieurs fois (la variable qui contrôle la boucle doit être modifiée en fonction du nombre de duplications). De plus, si la terminaison de la boucle est connue (i.e. le nombre d'itérations), elle peut être déroulée dans son intégralité.

— *Scindage de boucle* : décompose le corps de la boucle en plusieurs boucles ayant le même nombre d'itérations. Cependant, les relations entre les instructions du corps de la boucle doivent être prises en considération. Celui-ci doit être parallélisable.

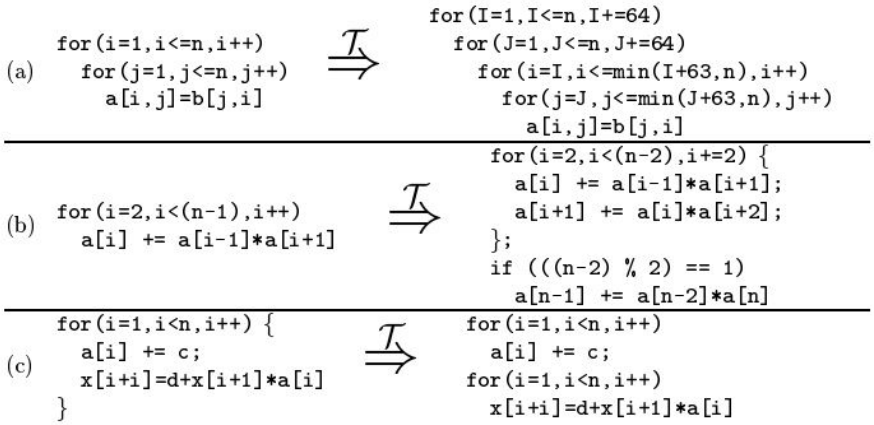

Fig 2.5 – Exemples de transformations de boucles. (a) découpage, (b) déroulage, (c) scindage (Extraite de [CTL97])

C. **Transformations des méthodes**

— *Inlining de méthodes* : remplace un appel à une fonction par son corps. Cela a l'avantage de modifier à la fois le GFC de la fonction appelante, et le graphe d'appels du programme.

— *Outlining de méthodes* : remplace un fragment de code particulier dans une fonction f par un appel à une nouvelle fonction g le contenant. Ceci change le GFC de la fonction f et ajoute un nouveau nœud g ainsi qu'un arc de f à g dans le graphe d'appels du programme.

— *Entrelacement de méthodes* : fusionne deux méthodes distinctes en une seule méthode (leurs corps et listes des paramètres) et rajoute un paramètre supplémentaire pour distinguer entre les appels aux deux méthodes originales.

— *Clonage de méthodes* : permet de créer plusieurs copies d'une même méthode en appliquant différentes techniques d'obfuscation. Chaque appel de la méthode originale est remplacé par un appel à l'un de ses clones.

D. **Parallélisation**

En principe elle est utilisée afin d'augmenter la performance d'un programme. Cependant, elle peut également servir pour cacher le flot de contrôle par la création des processus qui exécutent des tâches inutiles, ou bien la séparation d'une suite séquentielle de code en plusieurs parties qui peuvent être exécutées en parallèle. Pour cette seconde technique, nous avons deux cas possibles :

— Si les données sont indépendantes la parallélisation est simple (appels à des bibliothèques spéciales du langage utilisé).

— Si les données sont dépendantes, il faut utiliser la notion de concurrence entre les processus. Le nouveau programme sera exécuté séquentiellement mais le flot de contrôle passera d'un processus à un autre. L'avantage de cette technique est qu'elle rend l'analyse statique plus difficile car le nombre de chemins d'exécution possibles augmente exponentiellement avec celui de processus utilisés. Pour une analyse dynamique, cela est également plus complexe car l'attaquant doit analyser plusieurs processus en même temps.

E. **Insertion de code inutile/mort**

Comme indiqué précédemment, l'utilisation des prédicats opaques permet de créer de fausses branches dans un GFC. La complexité de cette technique repose sur la difficulté à déduire la valeur

du prédicat utilisé. Pour une séquence de blocs de base S qui est décomposée en deux sous-séquences $S1$ et $S2$, Collberg et al. [CTL97] exposent trois cas possibles (figure 2.6) :

— Cas 1 (figure 2.6-a) : mettre $S2$ dans une branche conditionnelle accessible après l'évaluation d'un prédicat opaque P^T (constitue le code inutile) qui est toujours évalué à vrai.

— Cas 2 (figure 2.6-b) : dupliquer $S2$ et obfusquer les deux copies séparemment en utilisant des techniques différentes (τ et $\tau\prime$) de telle façon à ce qu'un attaquant ne puisse pas déduire que les deux codes accomplissent les mêmes tâches. Autrement dit, quelle que soit la branche suivie après l'évaluation du prédicat opaque $P^?$, nous aurons le même résultat.

— Cas 3 (figure 2.6-c) : dupliquer $S2$ en $S2\prime$ et introduire une erreur dans l'une des deux versions. Par la suite, utiliser un prédicat opaque P^T ou P^F pour s'assurer qu'uniquement la version sans erreurs sera exécutée. L'autre version (celle avec erreurs) constitue un code mort.

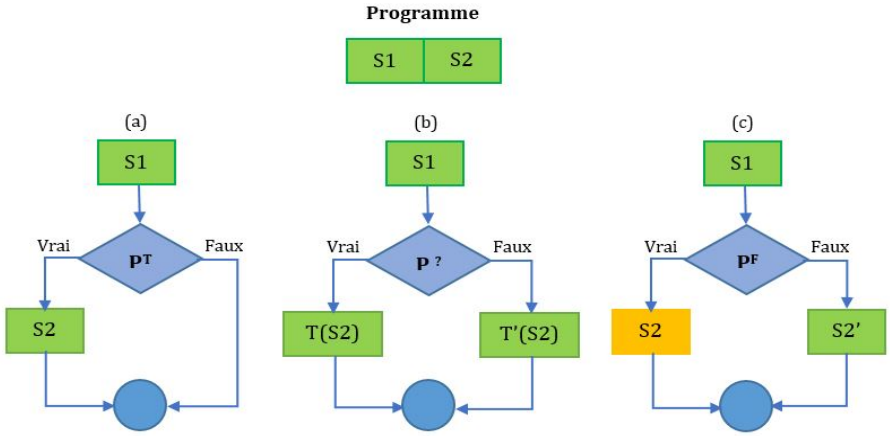

Fig 2.6 – Exemples d'insertion de code inutile/mort (adaptée de [CTL97])

2.5.4 Transformations préventives

Contrairement aux techniques des trois catégories précédentes, les transformations préventives visent à empêcher les décompilateurs et les déobfuscateurs de fonctionner correctement [Low98]. Elles peuvent être :

— Des transformations préventives *inhérentes* qui rendent des techniques automatiques connues de désobfuscation plus difficiles à appliquer.

— Des transformations préventives *ciblées* qui sont conçues pour contourner des outils spécifiques d'analyse.

2.6 Les techniques d'obfuscation appliquées aux codes malveillants

Les techniques d'obfuscation sont utilisées par les auteurs de malware dans le but de contourner le processus de détection. Cela se fait en dissimulant le code malveillant. Le camouflage dans les logiciels malveillants a suivi une croissance exponentielle au fil des années à partir du simple chiffrement aux malware polymorphes et métamorphes [GBS14, SGBS15]. Ces deux derniers types reposent sur des techniques qui modifient la signature de leur code à chaque génération [BS05] (i.e. suite à une nouvelle infection).

A chaque infection, un malware polymorphe chiffre son corps et change sa routine de déchiffrement, à l'aide d'un moteur de mutation [BS05]. Donc, il se compose de deux parties : un corps (partie constante) et une routine de déchiffrement (partie variable). Le fonctionnement d'un malware polymorphe est illustré par la figure 2.7.

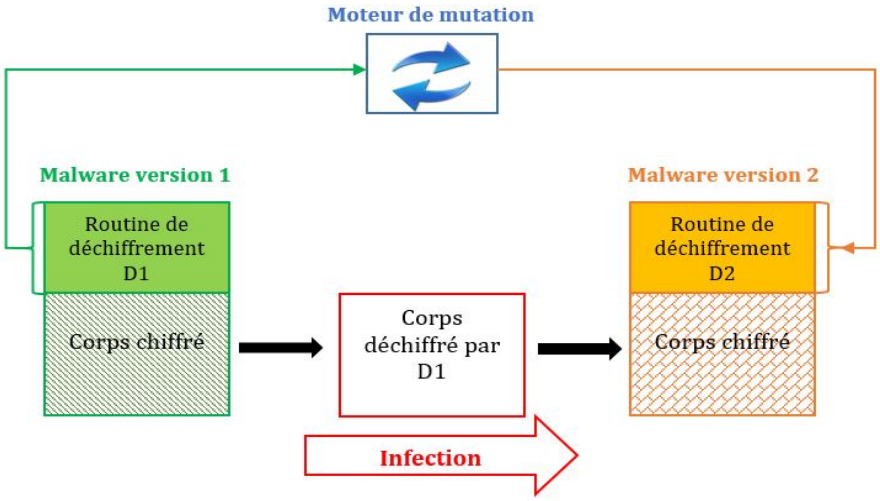

Fig 2.7 – Principe de fonctionnement d'un malware polymorphe

Un malware métamorphe est dit « polymorphe de corps », c'est-à-dire qu'au lieu de générer une nouvelle routine de déchiffrement (comme c'est le cas pour un malware polymorphe), un nouveau corps est créé, suite à chaque infection, tout en préservant ses fonctionnalités [SS14]. Le fonctionnement d'un malware métamorphe est illustré par la figure 2.8.

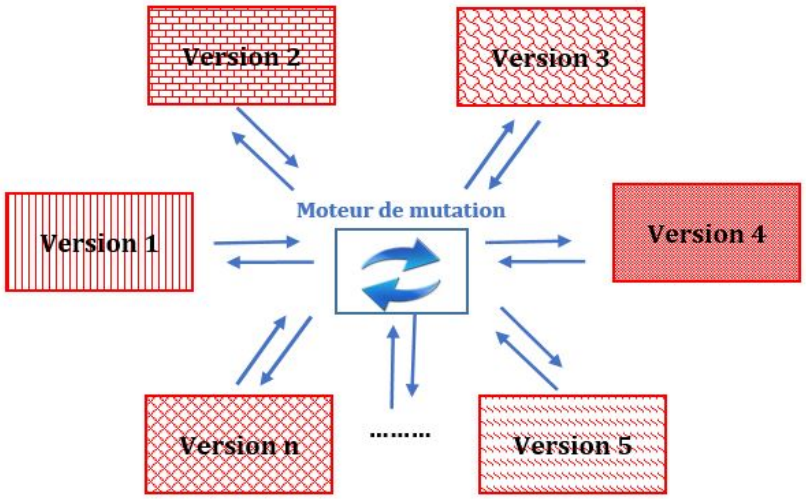

Fig 2.8 – Principe de fonctionnement d'un malware métamorphe

Les malware polymorphes et métamorphes utilisent tous les deux les techniques d'obfuscation du code [RMI12]. Le premier afin de muter sa routine de déchiffrement pour construire une nouvelle version et le second pour muter tout son corps et apparaître sous une nouvelle forme. Dans ce qui suit, nous allons présenter les techniques les plus courantes, parmi tant d'autres, dans la littérature utilisées pour l'obfuscation des malware. Plus de détails et d'exemples pratiques sont présentés dans [BM08, YY10, LS11, RMI12, SGBS15, FFL+16, SS18].

2.6.1 Insertion de code inutile

Elle consiste à placer des instructions inefficaces dans la structure du code afin de changer son apparence sans affecter son comportement. Ces nouvelles instructions peuvent être par exemple [BM08, BCD08] :

— Une instruction qui ne change pas le contenu des registres du CPU ou de la mémoire et qui est équivalente à une instruction `nop` (pour no-operation). Des exemples d'une telle instruction sont donnés dans le tableau 2.1.

Instruction	**Explication**
`add Reg, 0`	rajouter 0 à un registre
`mov Reg, Reg`	affecter la valeur contenue dans un registre à ce même registre
`or Reg, 0`	affecter à un registre le résultat d'un OU logique de sa valeur avec la valeur 0

Tab 2.1 – Exemples de code inutile (adapté de [BM08])

— Une instruction qui change probablement l'état de la mémoire ou les registres du CPU, mais son effet est annulé par une autre instruction avant d'affecter le résultat du programme. Deux exemples de combinaisons possibles sont donnés dans le tableau 2.2.

Exemple	**Explication**

`push CX`
`...`
`pop CX`
	La valeur de CX, qui est empilée au départ, doit être dépilée avant avoir un effet sur la pile

`inc AX`
`...`
`sub AX,1`
	La valeur de AX reste inchangée si elle n'est pas modifiée entre son incrémentation et sa décrémentation

Tab 2.2 – Exemples de code inutile (adapté de [RM11])

2.6.2 Insertion de code mort

C'est l'insertion de blocs de code inaccessibles et donc qui ne peuvent jamais être exécutés. L'inclusion d'un tel code dans un programme augmente la quantité de code qui doit être analysé. Ce qui rend son analyse plus coûteuse en termes de temps de calcul [FFL+16]. Pour rendre l'identification du code mort plus difficile, les prédicats opaques (voir section 2.5.3.3) peuvent être utilisés.

2.6.3 Substitution d'instructions

Ceci consiste au remplacement de certaines instructions du code initial avec d'autres instructions qui sont équivalents (en appliquant des règles de réécriture de code). Comme cette technique nécessite l'utilisation d'une librairie d'instructions équivalentes (qui est généralement non disponible), la signature du code original peut grandement changer [BS05]. Le tableau 2.3 présente quelques exemples de telles transformations.

2.6.4 Substitution de registres

Elle vise à changer les registres utilisés dans les différentes versions du code. Le fonctionnement global est préservé tandis que la structure du code est changée. Le tableau 2.4 présente un exemple de

Instruction	Instructions équivalentes
mov Reg, 0	`xor Reg, Reg`
	`and Reg, 0`
	`sub Reg, Reg`
mov Reg, val	`push val` `pop Reg`

OP reg, Reg2	`mov Mem, Reg`
	`OP Mem, Reg2`
	`mov Reg, Mem`

Tab 2.3 – Exemples de combinaisons de substitution d'instructions possibles (adapté de [Bor11, RMI12])

deux programmes où les registres edx, edi, esi, eax et edx du programme 1 sont substitués respectivement par les registres eax, ebx, edx, edi et eax du programme 2.

Code 1	Code 2
`pop edx`	`pop eax`
`mov edi, 0x04`	`mov ebx, 0x04`
`mov esi, 0x12`	`mov edx, 0x12`
`mov eax, 0x0C`	`mov edi, 0x0C`
`add edx, 0x88`	`add eax, 0x88`

Tab 2.4 – Exemple de substitution de registres (extrait de [Bor11])

2.6.5 Transposition de code

Elle réorganise l'agencement des instructions du code original sans avoir des effets sur son comportement [YY10]. Grâce à ce processus de réorganisation, lorsque différentes combinaisons d'instructions sont appliquées, la structure du code semble différente dans diverses générations [RMI12]. Les nouvelles générations peuvent être créées, par exemple :

— En choisissant et en réorganisant les instructions indépendantes qui n'ont aucun impact les unes sur les autres (tableau 2.5).

Code 1	Code 2
`sub eax, ebx`	`mov ecx, 0C`
`mov ecx, 0C`	`pop edx`
`pop edx`	`sub eax, ebx`

Tab 2.5 – Exemple de permutation d'instructions indépendantes

— En réorganisant les instructions du programme tout en conservant le flot d'exécution moyennant les branchements conditionnels ou inconditionnels (figure 2.9).

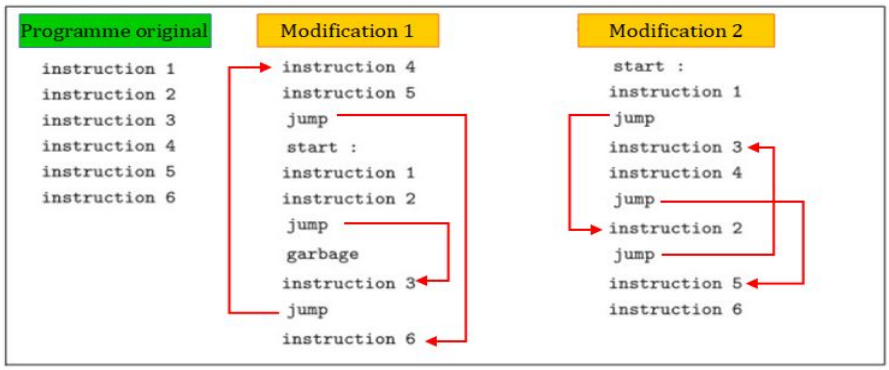

Fig 2.9 – Exemple de modification du flot de contrôle (adaptée de [BM08])

2.7 Conclusion

Indépendamment de l'utilisation des techniques d'obfuscation (par les développeurs ou les attaquants), l'objectif est de dissimuler un code original dans le but de le rendre incompréhensible pour un lecteur humain ou un processus d'analyse automatique. Dans ce chapitre, nous avons passé en revue les différentes techniques d'obfuscation de code (sain ou malveillant). Elles peuvent être utilisées individuellement ou en combinaison afin d'augmenter le niveau de difficulté voulu et ainsi avoir un code final plus difficile à comprendre et analyser tout en préservant ses fonctionnalités.

CHAPITRE 3

Les attaques par injection de fautes

Sommaire

Les *attaques par injection de fautes* (ou *Attaques en fautes*[1], en anglais *Fault Attacks*) visent à perturber le fonctionnement normal d'un système (modification du comportement) en introduisant des modifications physiques dans son environnement. Le but étant d'introduire des fautes durant l'exécution d'un programme afin d'obtenir des informations sensibles comme les clés cryptographiques ou d'effectuer des traitements non autorisés (par exemple éviter le test d'un code PIN) [Sér10].
Dans le cadre de cette thèse, nous nous intéressons aux attaques par injection de fautes comme moyen d'activation des codes malveillants contre les cartes à puce (des exemples d'attaques existantes sont donnés dans la section 3.6). En effet, le système en question est un ensemble de circuits électroniques dont l'environnement peut être perturbé moyennant différentes techniques (section 3.2) et induisant plusieurs effets sur le comportement normal de la cible (section 3.3) qui sont classés selon différents modèles de fautes (section 3.4). Dans ce qui suit, nous allons présenter chacun de ces points afin d'avoir une vue générale sur les attaques par injection de fautes.

3.1 L'injection de fautes : origines

Les origines de l'injection des fautes sont liées au domaine aérospatial. Ceci remonte à 1979, où Ziegler et Lanford [ZL79] ont constaté que les rayons cosmiques, qui sont des particules énergétiques

1. Dans la suite du document, nous utiliserons indifféremment les deux appellations : attaques par injection de fautes ou attaques en fautes.

provenant de l'espace, pourraient perturber le comportement de certains appareils électroniques présents dans les avions ou les véhicules spatiaux. Notamment, ils ont observé que ces particules pouvaient perturber la mémoire des appareils et conduire à la corruption de certaines exécutions du programme [Chr13].
En 1997, le point de départ officiel du domaine des attaques par perturbation est marqué avec les travaux de Boneh et al. [BDL97]. Les auteurs démontrent qu'une unique perturbation durant un calcul cryptographique permet de retrouver une clé privée RSA et de casser d'autres algorithmes cryptographiques. Par la suite, l'impact des phénomènes physiques sur les systèmes embarqués a été largement étudié par la communauté scientifique, avec un intérêt particulier aux systèmes sécurisés (spécialement les cartes à puces) [BDH11]. En conséquence, de nouvelles attaques ont été mises en œuvre et des contre-mesures spécifiques ont été conçues pour y faire face.

3.2 Les techniques d'injection de fautes

Dans la pratique, il existe différentes façons pour injecter une faute. Dans cette section, nous présentons brièvement les techniques qui sont le plus souvent citées dans la littérature. Plus de détails peuvent être trouvés dans [BECN+06, KSV13, PBR17, YSW18].

Perturbation de l'alimentation

Appelée *attaque électrique* [ABF+03, SGD08, BBPP09, BBBP13, TSW16], elle consiste à faire varier brusquement la tension d'entrée de l'alimentation de la puce (des courtes impulsions électriques sur l'alimentation, des pics d'alimentation ou des micro-coupures) afin de perturber l'exécution de certaines opérations.

Perturbation de l'horloge

Une attaque par variation de la fréquence de l'horloge [GSD+08, Ta10, BGV11, KH14] consiste à faire varier la vitesse de l'horloge hors des limites autorisées par la carte afin d'introduire des fautes au niveau du microprocesseur (une mauvaise lecture de données ou un saut de l'instruction en cours d'exécution).

Température

La modification de la température (en chauffant ou en refroidissant le composant) [GA03, HS13] entraine la modification du contenu des cellules de la mémoire de manière aléatoire ainsi qu'une perturbation des lectures/écritures de ces dernières.

Émissions lumineuses

Appelée *attaque optique* [AS02, SA02, Sko10, VTM+18], elle consiste à utiliser l'énergie d'une émission lumineuse (laser, flash, infrarouge, lumière blanche, *etc.*), concentrée sur la surface de la puce, pour perturber le silicium du composant. Ainsi, il est possible d'apporter suffisamment d'énergie à une cellule mémoire pour lui faire changer son contenu.

Émissions électromagnétiques

Une attaque par perturbation électromagnétique [QS02, SH07a] consiste à émettre une forte pulsation magnétique près d'une cellule mémoire. Le champ magnétique crée des courants locaux

à la surface du composant, pouvant ainsi générer une modification du contenu des cellules de la mémoire.

Le tableau 3.1 récapitule les techniques d'injection de fautes décrites précédemment. Pour chaque technique d'injection de faute, le tableau fournit la précision spatiale et temporelle (voir section 3.4.3) ainsi que le coût matériel nécessaire à sa réalisation. Ces différentes techniques peuvent être combinées afin d'augmenter la puissance d'une l'attaque en fautes [KHEB14, KH14].

Technique d'injection	**Précision spatiale**	**Précision temporelle**	**Coût matériel**
Surcadençage (overclocking)	faible(globale)	faible(globale)	faible
Glitch de l'horloge	faible(globale)	élevée(locale)	faible
Sous-alimentation	faible(globale)	faible(globale)	faible
Glitch de la tension	faible(globale)	élevée(locale)	faible
Surchauffage	faible(globale)	faible(globale)	faible
Impulsion lumineuse	moyenne(locale)	moyenne(locale)	faible
Impulsion laser	élevée(locale)	élevée(locale)	élevé
Impulsion électro-magnétique	moyenne(locale)	élevée(locale)	élevé

Tab 3.1 – Les techniques d'injection de fautes (adapté de [YSW18])

3.3 Les conséquences des injections de fautes

Une injection de faute peut provoquer différents effets qui sont plus au moins graves en fonction des conséquences engendrées sur la cible. Cette dernière peut être des cellules de la mémoire, un bus de mémoire, quelques registres du microprocesseur, ou un coprocesseur cryptographique. D'après [Sér10] et [Chr13], nous pouvons donner les exemples de perturbations suivantes :

- *Perturbation du microprocesseur :*
  - Si les registres de calcul sont perturbés, alors il est probable que les résultats intermédiaires soient erronés.
  - Si les registres spéciaux sont perturbés, des branchements ou des instructions peuvent être injectés dans le code. Quand la modification concerne le registre du compteur ordinal (en anglais « *program counter register* »), un saut dans le code est réalisé. Une autre cible possible est le pointeur de la pile (en anglais « *stack pointer* »), en modifiant sa valeur une zone mémoire différente de celle qui devait être exécutée serait considérée comme la pile, ce qui peut conduire à l'exécution de mauvaises branches à la fin de la fonction en cours.
- *Perturbation de la mémoire :*
  - Dans l'EEPROM ou la mémoire flash où peuvent être stockés le code des applications, les clés cryptographiques, ou même le code PIN de l'utilisateur.
  - Dans la RAM, où la structure des objets temporaires manipulés peut être modifiée.
- *Perturbation du bus :*
  - Pendant la lecture/écriture des données, et donc la variable en question est perturbée.
  - Durant la lecture d'une instruction, et donc une autre instruction est exécutée à sa place (par exemple : sauter l'exécution d'une fonction quand il s'agit d'une branche, exécuter une instruction aléatoire, fausser un calcul en réalisant un `add` à la place d'un `xor` par exemple, *etc.*)

En résumé, une attaque en faute entraîne des modifications du code, des perturbations dans le flot de contrôle, le flot de données, ou une modification des données [Sér10]. Des exemples de telles pertubations sont [YSW18] : saut d'une instruction [BTG10, DMM+13] (i.e. éviter l'exécution d'une

instruction spécifique), saut de plusieurs instructions [RNR+15, NHH+17], modification d'instruction [TSW16, BGV11], changement du résultat d'une branche conditionnelle [VF10, PMPD14], et altération du compteur de boucle [CT05, DMN+12].

3.4 Les modèles de fautes

Dans le but d'évaluer les conséquences possibles d'une attaque par injection de fautes et d'empêcher ainsi sa survenue, il est nécessaire de modéliser la perturbation qui pourrait être provoquée. Certaines de ces attaques sont très génériques et nécessitent uniquement une perturbation du calcul pendant l'exécution, alors que d'autres nécessitent que certaines hypothèses soient vérifiées concernant le type de faute provoqué [Chr13]. Ceci, correspond à un *modèle de fautes*. Avant d'aborder les différents modèles de fautes existants dans la littérature, nous allons présenter trois concepts importants qui y sont liés, à savoir : la persistance et les types des fautes ainsi que les paramètres d'un modèle de fautes.

3.4.1 Persistance des fautes

Les fautes introduites par une attaque en fautes peuvent être : permanentes ou transientes.

— *Fautes permanentes* : Elles consistent à changer la valeur d'une cellule mémoire définitivement. Cela peut concerner soit des variables ou bien du code. Ces modifications peuvent être extrêmement efficaces pour un attaquant, en particulier lorsqu'elles sont liées aux objets sensibles de la carte, tels qu'un code PIN ou une clé cryptographique.

— *Fautes transientes* : Elles consistent à changer la valeur d'une cellule mémoire temporairement. Elles sont obtenues lorsque l'exécution du code ou un calcul sont perturbés. Après un certain temps, l'effet de la faute disparait et la variable erronée reprend sa valeur initiale.

3.4.2 Type de fautes

La caractérisation du type de fautes est faite selon les modifications que les attaques provoquent à un seul bit. Les effets d'une injection de fautes, sur un bit, peuvent être :

— *Bit flip* : la valeur du bit est inversée

— *Bit set* : le bit est forcé à 1

— *Bit reset* : le bit est forcé à 0

— *Collage (stuck-at)* : le bit est maintenu à sa valeur précédente

— *Aléatoire* : le bit prend une valeur aléatoire

3.4.3 Paramètres d'un modèle de fautes

Les modèles de fautes communément considérés dans la littérature dépendent principalement des critères suivants [Sér10] :

— La *précision* est la quantité d'information modifiée par l'attaque. Il peut s'agir d'un bit, d'un octet ou d'une valeur variable. Il faudrait noter que plus la carte contient des mécanismes de protection, plus il est difficile de modifier une zone suffisamment petite et précise de la mémoire.

— La *localisation spatiale* de la faute à la surface de la puce indique à quel endroit de la carte la faute est injectée. Ceci nécessite une distinction des différents types de mémoire.

— La *fenêtre temporelle* pendant laquelle l'attaque s'effectue. En effet, une attaque réussie doit pouvoir être synchronisée avec une instruction ou alors avec la manipulation d'une donnée importante.

— Le *type de faute* qui est l'état de la zone mémoire modifiée après l'attaque (section 3.4.2). En fonction du type de la mémoire, une attaque peut changer l'information vers une valeur donnée (mémoires non chiffrées) ou aléatoire si cette valeur est soumise à un aléa (mémoires chiffrées).

3.4.4 Les modèles

Les modèles de fautes existants ont déjà été discutés en détail dans [Ott05, Gir07]. Le tableau 3.2 résume les différents modèles de faute existants et qui sont donnés par ordre décroissant en termes de difficulté de réalisation. Pour les critères de fenêtre temporelle et de localisation, l'attaquant a trois niveaux de contrôle [Sér10] :

— *Total* : il a un contrôle total sur le critère.
— *Moyen* : il a un contrôle peu précis sur le critère.
— *Aucun* : il n'a aucun contrôle du critère.

Modèle de faute	Précision	Localisation	Timing	Type de fautes
Erreur de bit précis	bit	total	total	bsr [2]
Erreur d'octet précis	octet	total	total	bsr, aléatoire
Erreur d'octet inconnu	octet	moyen	moyen	bsr, aléatoire
Erreur aléatoire	variable	aucun	aucun	aléatoire

Tab 3.2 – Les modèles de fautes existants (Extrait de [Sér10])

Une attaque utilisant le modèle d'erreur de bit précis a été décrite par Skorobatov et Anderson dans [SA02]. Cependant, ce modèle n'est pas réaliste sur les cartes à puce actuelles en raison de l'implémentation de la sécurité matérielle sur la mémoire des composants modernes (par exemple, correction d'erreur ou chiffrement de la mémoire). Un modèle largement accepté correspond au modèle d'erreur d'octet précis où un attaquant peut changer un octet à un temps précis et synchronisé [VF10]. Dans la pratique, un attaquant injecte physiquement de l'énergie dans une cellule mémoire pour changer son état. Ainsi, en fonction de la technologie sous-jacente, la cellule mémoire cible prend physiquement la valeur `0x00`, `0xFF` ou une valeur aléatoire si les mémoires sont chiffrées (cette valeur dépend des données, de l'adresse et d'une clé de chiffrement) [BLM+11].

3.5 Qu'est ce qu'un mutant ?

3.5.1 Définition

La génération et la détection de mutants est un champ de recherche introduit initialement par [VF10, BTG10]. Dans le contexte Java Card , d'après Séré [Sér10], « *Une application mutante est une application Java Card qui a subi une modification due à une attaque (en faute par exemple) et qui continue d'avoir du sens pour la machine virtuelle lors de l'interprétation. Ce qui veut dire que cette application ne cause aucun plantage de la machine virtuelle et qu'elle arrive à passer outre les mesures de sécurité qui lui sont intégrées* ».

3.5.2 Exemple d'une application mutante

La définition donnée ci-dessus (section 3.5.1) est illustrée à travers l'exemple présenté dans cette section et qui est extrait de [AKSICL11]. Il s'agit d'une méthode de débit issue d'une application de gestion d'un porte-monnaie électronique (Wallet Java Card applet). Cette méthode vérifie que

2. bit set or reset

le code PIN de l'utilisateur est correct avant d'effectuer l'opération de diminution du solde de son porte-monnaie. Un extrait du code source de cette méthode accompagné du bytecode correspondant est donné dans la figure 3.1.

Bytecode	Octets	Source Java
00 : aload_0	00 : 18	private void debit (APDU apdu) {
01 : getfield #4	01 : 83 00 04	
04 : invokevirtual #18	04 : 8B 00 23	
07 : ifeq 59	07 : 60 00 3B	if (pin.isValidated()) {
10 : ...	10 : ...	// make the debit operation
...	...	} else {
59 : sipush 25345	59 : 13 63 01	ISOException.throwIt (
63 : invokestatic #13	63 : 8D 00 0D	SW_PIN_VERIFICATION_REQUIRED);
66 : return	66 : 7A	}

Fig 3.1 – Représentation de la méthode debit() avant l'attaque (Extraite de [Sér10])

Un attaquant vise à éviter la vérification du code PIN. Pour cela, il doit réaliser une injection de faute ciblant l'instruction de test conditionnel i.e. la cellule contenant l'opcode `ifeq` (l'octet dont la valeur correspond à `0x60`). Le résultat obtenu est présenté dans la figure 3.2.

Bytecode	Octets	Source Java
00 : aload_0	00 : 18	private void debit (APDU apdu) {
01 : getfield #4	01 : 83 00 04	
04 : invokevirtual #18	04 : 8B 00 23	
07 : nop	07 : 00	~~if (pin.isValidated())~~ {
08 : nop	08 : 00	
09 : pop	09 : 3B	
10 : ...	10 : ...	// make the debit operation
...	...	} else {
59 : sipush 25345	59 : 13 63 01	ISOException.throwIt (
63 : invokestatic #13	63 : 8D 00 0D	SW_PIN_VERIFICATION_REQUIRED);
66 : return	66 : 7A	}

Fig 3.2 – Représentation de la méthode debit() après l'attaque (Extraite de [Sér10])

Suite à l'injection de la faute (suivant un modèle de fautes erreur d'octet précis), l'instruction `ifeq` change pour une instruction `nop` (dont la valeur est `0x00`). Ainsi, la partie haute de l'adresse qui correspond au premier opérande de `ifeq` est interprétée comme étant une instruction (`nop` dont la valeur est `0x00`) ; il en est de même pour la partie basse de l'adresse correspondant au deuxième opérande qui est interprétée comme étant une instruction `pop` (dont la valeur est `0x3B`). Par la suite, la machine virtuelle continuera d'interpréter le code de façon normale (le flot initial est retrouvé). Donc, la vérification du code PIN est contournée, l'opération de débit est effectuée et une exception est levée mais elle intervient trop tard car l'attaquant a déjà atteint son but. L'attaque ne peut réussir qu'avec une injection de fautes correctement synchronisée avec l'exécution d'une instruction précise.

3.6 Les attaques activables par attaques en faute : attaques combinées

Bien que les attaques en fautes soient utilisées dans l'analyse cryptographique, elles peuvent également être utilisées pour déclencher des attaques logiques sur les processeurs d'ordre général [MDH+13, RNR+15, TSW16, NHH+17, KMW17, BLLL18] et les cartes à puce comme cas particulier.

Comme nous nous intéressons à ces dernières (cible de notre travail), des exemples d'attaques combinées exploitant des injections de fautes et ciblant les différents éléments du système sont brièvement présentées ci-dessous.

Contourner le pare-feu

Vetillard et Ferrari ont présenté dans [VF10] un travail théorique portant sur le développement d'attaques combinant des attaques logiques à d'autres physiques. Les auteurs ont exposé un scénario, testé sur des cartes classiques Java Card 2.2, relativement simple par rapport à d'autres cas possibles. L'idée est de forger une référence, récupérée à l'aide d'une technique de dump de la mémoire, dans une application valide. Par la suite, réaliser une injection de l'instruction `nop` (i.e. changer la valeur d'une cellule mémoire en `0x00`) afin d'éviter un test d'accès aux références (i.e. contourner le pare-feu). Cette attaque a permis de récupérer les clés secrètes, qui sont censées être confidentielles, possédées par une autre application.

Contourner le BCV

Barbu et al. ont présenté dans [BTG10] la première attaque pratique combinant une injection de fautes avec une attaque logique. Deux cas d'étude, réalisés sur des cartes Java Card 3.0, ont été présentés. Le premier cas montre comment introduire un code mal-formé bien qu'un vérifieur de bytecode embarqué soit présent dans la carte. Alors que le second exemple expose comment transformer une méthode quelconque présente dans la carte en un code malicieux. L'attaque consiste à installer une applet correcte (vérifiée par le BCV) contenant une conversion de type non autorisée entre deux objets différents. Si une attaque en faute cible l'instruction de contrôle de type (`checkcast`) de telle sorte qu'elle ne soit pas exécutée, cette applet devient hostile et peut alors exécuter n'importe quel shellcode. Ce type d'attaque exploite une nouvelle méthode pour exécuter des instructions illégales où les niveaux physiques et logiques sont perturbés. Cette méthode ne réussit que sur certaines cartes (les autres cartes semblent ne pas y être sensibles).

EMAN4 : perturbation du flot de contrôle

Bouffard et al. [BICL11], ont proposé une attaque pour perturber le graphe du flot de contrôle d'une applet moyennant une injection de fautes. Les auteurs ont décrit l'attaque sur une boucle `for`, avec une possibilité d'extension à d'autres instructions conditionnelles. La spécification Java Card [Ora15b] définit deux instructions pour créer une branche à la fin d'une boucle : `goto` (prend un décalage d'un octet) et `goto w` (prend un décalage de deux octets). Le choix des auteurs a porté sur l'instruction `goto _w`. L'injection de la faute a ciblé le premier paramètre de cette instruction. Sa valeur change de `0xFF` à `0x00`. Ainsi, au lieu de faire un saut arrière (une autre itération de la boucle) l'instruction effectue un saut avant afin de pointer sur un tableau contenant un shellcode.

Activation de code mort

Bouffard et Lanet dans [BL15], proposent une attaque qui permet d'activer un fragment de code mort (i.e. un code non atteignable dans le graphe de flot de contrôle) en contournant le BCV. Après étude du comportement de ce dernier envers un code mort, ils ont trouvé que la destination du saut est toujours contrôlée mais la sémantique du code n'est pas vérifiée. Ceci dit, afin d'exploiter cette faiblesse du BCV, les auteurs ont proposé une extension de leur précédente attaque [BICL11]. Le code qu'ils ont utilisé est présenté dans le listing 3.1.

```
1  void abuseBCV () {
2  04 // flags : 0 max_stack : 4
3  03 // nargs : 0 max_locals : 3
4  L0: jsr L1
5  sspush 0xCAFE
6  sreturn
7  sspush 0xBEEF
8  sreturn
9  astore_3 //stocker l'adresse de retour
10 L1: //
11   ...
12 sspush VALUE_1
13 sspush VALUE_2
14 if_scmpeq_w 0 xFF05 // => L1
15 return
16 //************** Code mort ****************
17 sinc 0x03 , 0x04
18 ret 0x03
19 //************** Code mort ****************
20 }
```

Listing 3.1 – Exemple de code mort non vérifié par le BCV (Extrait de [Bou14])

Les lignes 17 et 18 constituent le fragment de code non atteignable à activer. Pour cela, l'injection de la faute cible le premier paramètre de l'instruction `if_scmpeq` (ligne 14) ce qui va transformer la valeur `0xFF05` à `0x0005`. Le saut sera exécuté de telle sorte qu'il atteindra le code mort. Dans cet exemple, c'est l'adresse de retour de la fonction qui sera modifiée (ligne 17) pour une redirection vers un shellcode.

Contourner le BCV : attaque persistante

Mesbah et al. [MML17] ont étudié le comportement du BCV à l'égard des codes morts et ont trouvé un moyen de le contourner. Ensuite, ils ont démontré comment cette brèche pourrait être utilisée pour accéder aux données système d'une frame (plus exactement l'adresse de retour d'une fonction), ceci dans le but d'activer ou modifier de manière persistante n'importe quel code. Pour cela, ils ont utilisé une approche en boîte blanche (basée sur les résultats d'un travail antérieur [MLM17]) associée à une attaque par injection de fautes dans le but de transformer un code bien-formé en un code mal-formé pendant l'exécution.

3.7 Conclusion

Dans ce chapitre, nous avons présenté les attaques par injection de fautes en tant que moyen d'activation de codes malveillants. En effet, avec l'avancement des recherches dans le sens de l'amélioration du processus de vérification des applications pour cartes ainsi que l'obligation de l'utilisation du vérifieur de bytecode embarqué dans les nouvelles versions de Java Card (à partir de la version 3.0), l'idée de combiner des attaques logiques et physiques est devenue la solution possible pour faire aboutir les attaques contre les cartes à puce.

Notre positionnement

Cette partie a été dédiée à la présentation du contexte de notre travail (les cartes à puce Java Card) ainsi que l'état de l'art relatif aux éléments clés liés à la problématique traitée qui s'inscrit dans le domaine de la sécurité des cartes à puce. Comme exposé, cette dernière peut être contournée de plusieurs façons :

- Des attaques matérielles qui manipulent la carte physiquement et nécessitent des moyens importants ;
- Des attaques logiques qui profitent du caractère de la plateforme Java Card (ouverte et multi-applicative) en chargeant des applications malicieuses bien-formées ou mal-formées. Ces attaques reposent sur deux hypothèses : l'attaquant dispose des clés de chargement des applications (chargement en post-issuance) et la carte ne dispose pas de vérifieur de bytecode embarqué. Mais avec les dernières versions de Java Card qui imposent l'utilisation de ce dernier, de telles attaques sont devenues très difficiles à réaliser.
- Des attaques combinées qui permettent de relaxer l'hypothèse précédente (avoir un BCV embarqué) et ainsi pouvoir charger un code malicieux (attaque logique) qui sera activé dans la carte via une attaque par injection de fautes (attaque physique).

Notre travail appartient à la catégorie des attaques combinées. A la différence des attaques présentées dans la section 3.6, nous ne cherchons pas à contourner le BCV embarqué vu que le code à charger peut passer l'étape de vérification sans qu'il ne soit détecté. En effet, dans notre cas le code malicieux sera dissimulé dans un code sain qui sera chargé sans poser de problèmes de vérification (parce qu'il est valide vis-à-vis de la spécification Java Card), une fois dans la carte il subit une injection de fautes ciblée qui va activer le code malicieux initialement caché. Autrement dit, nous cherchons à avoir un code avec deux sémantiques : avant et après l'injection de la faute. Les deux sont correctes i.e. respectent la spécification de la JCVM [Ora15b]. Un autre point important à souligner est qu'à la différence des attaques combinées existantes qui sont spécifiques, nous visons à mettre en œuvre une approche de construction de code malveillant indépendamment de son contenu i.e. le comportement hostile caché. Ce qui élargit le spectre des attaques possibles : une seule méthode de construction pour plusieurs attaques.

Comme les techniques d'obfuscation (chapitre 2), notre travail vise à dissimuler un code pour rendre sa compréhension et analyse difficile. Néanmoins, indifféremment de ces techniques, qui exigent la préservation de la sémantique du code caché, nous voudrions augmenter encore le niveau de difficulté en dissimulant même la sémantique du code malveillant afin qu'il apparaisse comme un code sain. D'après nos recherches, cette problématique n'a pas été traitée auparavant ni dans le domaine de la sécurité ni dans celui de l'ingénierie logicielle (construction de programmes). D'où l'originalité du présent travail : nous visons à construire un programme avec deux sémantiques correctes. Initialement

le comportement hostile (sémantique 1) est dissimulé dans un autre code (sémantique 2) pour être récupéré après l'injection d'une faute.

La deuxième partie de cette thèse sera consacrée pour répondre à la question :

« Comment construire un code malveillant activable par attaque en fautes pour un support d'exécution sécurisé ? »

Partant d'un code hostile, nous cherchons à construire du code autour afin de le dissimuler. Ceci revient à rajouter une ou plusieurs instructions avant son début de telle façon à ce que ce dernier soit fondu dans le reste du nouveau programme. Autrement dit, dans le programme résultant, le code hostile initial, sa structure, ne pourra pas être repéré. La difficulté consiste à trouver la bonne instruction à rajouter et qui une fois la faute injectée donnera le comportement voulu. De plus, nous devons garantir que l'insertion de ces instructions se fasse en respectant les contraintes qui seront vérifiées ultérieurement par le vérifieur de bytecode. Vu la difficulté du problème principal, nous avons jugé plus judicieux de le répartir en deux sous-problèmes pour lesquels nous avons apporté deux solutions complémentaires :

- Sous-problème 1 : Concerne la construction d'une séquence de code en faisant abstraction à l'injection de fautes. Ceci revient à trouver une séquence d'instructions liant deux fragments de code donnés tout en respectant un ensemble de contraintes données. La modélisation de ce problème ainsi que la solution apportée feront l'objet du chapitre 4.
- Sous-problème 2 : Concerne ce que nous avons appelé la désynchronisation du code. En effet, l'ajout des instructions avant le début du code hostile n'est pas sans conséquences. Ceci conduit à un décalage du code initial et donc produit un nouveau code ayant une sémantique différente. Plusieurs cas possibles peuvent être distingués. L'étude du mécanisme de désynchronisation est donnée dans le chapitre 5.

Pour résumer, notre travail est basé sur les hypothèses suivantes :

- Le support d'exécution est une carte à puce.
- La cible est une application Java Card.
- La construction du code est faite au niveau bytecode.
- L'injection de faute est le déclencheur de notre code malveillant. Nous nous sommes intéressés uniquement à l'effet de la faute et non pas au moyen physique utilisé pour réaliser l'injection (nous ne traitons pas le comment).
- Le modèle de faute choisi est un modèle d'erreur d'octet précis avec une mémoire non chiffrée. Quand la faute est injectée, l'instruction stockée dans la cellule mémoire cible prend la valeur `0x00` (correspond à une instruction `nop`)

Deuxième partie

Contributions

CHAPITRE 4

Construction de séquences de code par parcours d'arbre

Sommaire

Dans ce chapitre, nous allons répondre à notre premier sous-problème : la construction d'une séquence de code liant deux états mémoire. D'abord, nous allons présenter le cadre théorique de notre contribution et qui est fondé sur le domaine de la résolution de contraintes (section 4.2). En effet, nous allons montrer comment le problème traité peut se ramener à un problème de résolution de contraintes (section 4.3) puis donner les éléments nécessaires à sa modélisation (section 4.4). Par la suite, nous détaillerons l'approche que nous avons proposé pour la construction de nos séquences de code en se

basant sur un parcours d'arbre (section 4.5). A la fin, nous proposerons une optimisation de cette approche en introduisant des heuristiques de prise de décision (section 4.6).

4.1 Le problème de construction de séquences de code

La première étape pour cacher un code hostile dans un code sain consiste à trouver, parmi l'ensemble des instructions bytecode existantes, une séquence d'instructions à ajouter dans le but de lier deux instructions données. La première étant le début du code hostile à cacher et la seconde représente la fin d'un fragment de code inoffensif. Cette construction se fait dans le sens inverse de l'exécution (vers l'arrière). De plus, nous devons nous assurer que l'insertion de ces instructions respecte un ensemble de contraintes, qui seront vérifiées ultérieurement par le vérifieur de bytecode, afin d'obtenir un programme syntaxiquement et sémantiquement correct.

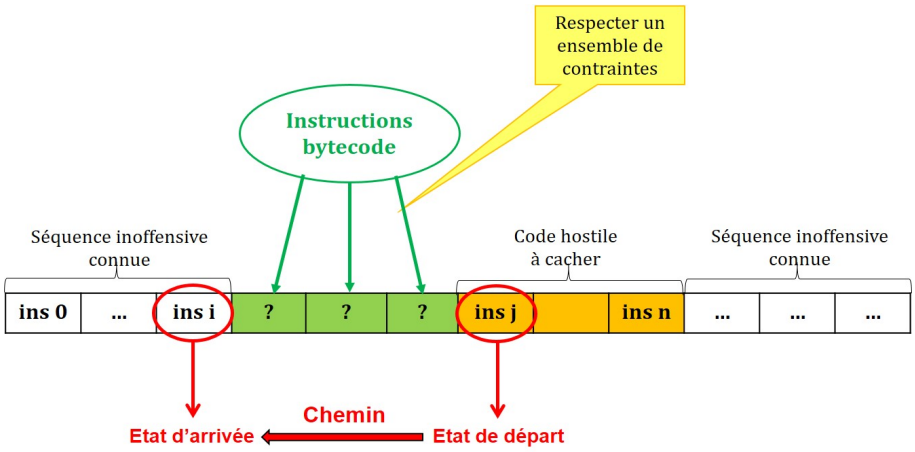

Fig 4.1 – Problème de construction de séquence de code

Lors de son exécution, chaque instruction bytecode modifie l'état de la mémoire d'une manière définie par sa sémantique. L'effet de l'exécution de chaque instruction peut être considéré comme étant une relation entre deux états de la mémoire : l'état de la mémoire avant l'exécution de l'instruction et l'état de la mémoire après son exécution. Cette relation se traduit notamment par des contraintes qui portent sur des éléments composant les états mémoire.

L'objectif de notre approche est donc, à partir d'un état mémoire de départ représentant le début du code hostile, d'insérer des instructions bytecode et de recalculer l'état mémoire précédent afin de converger vers l'état mémoire à rejoindre (la fin du code sain). La construction de la séquence doit résoudre deux problèmes : le choix d'une instruction parmi celles qui existent dans le langage Java Card et le calcul de l'état mémoire précédent cette instruction. L'objectif du choix de l'instruction est de se rapprocher de l'état mémoire d'arrivée tout en respectant les contraintes définies.

Pour ce faire, il fallait passer par les deux étapes suivantes qui sont détaillées dans les sections à venir :

— Modéliser le problème de la construction de séquences de code comme un problème de résolution de contraintes.
— Résoudre le problème modélisé en appliquant les techniques connues dans ce domaine avec adaptation à notre cas.

4.2 Les problèmes de satisfaction de contraintes (CSP) : représentation et résolution

La programmation par contraintes est un paradigme, puissant et bien étudié, utilisé pour résoudre des problèmes de recherche combinatoire. Il est appliqué avec succès à de nombreux problèmes connus, tels que l'ordonnancement, la planification, le routage des véhicules, les problèmes d'optimisation, la biologie moléculaire, l'allocation de ressources, synthèse de circuits électroniques, etc. Fondamentalement, un problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem en anglais) est un problème composé d'un ensemble fini de variables, chacune associée à un domaine fini, et un ensemble de contraintes qui limitent les valeurs que les variables peuvent prendre simultanément [Tsa95]. L'objectif est d'attribuer à chaque variable une valeur satisfaisant toutes les contraintes. Nous soulignons que dans le cadre de ce travail nous nous intéressons aux CSP discrets, c'est à dire aux CSP dont le domaine des variables consiste en un ensemble fini de valeurs. Les définitions des concepts introduits dans cette section sont données dans l'annexe B.

4.2.1 Définition d'un CSP

Plus formellement, un CSP est défini comme suit [Tsa95] :

Définition 4.1. ***(Problème de satisfaction de contraintes)*** *Un CSP est un triplet* (X, D, C)*, où :*

- $X = \{x_1, x_2, ..., x_n\}$ *est l'ensemble des variables ;*
- $D = \{d_{x_1}, d_{x_2}, ..., d_{x_n}\}$ *est l'ensemble des domaines. Chaque domaine* d_{x_i} *est un ensemble fini contenant les valeurs possibles de la variable* x_i *;*
- $C = \{c_1, c_2, ..., c_m\}$ *est l'ensemble des contraintes. Une contrainte est simplement une relation logique, elle concerne une ou plusieurs variables, qui limite les valeurs possibles que les variables peuvent prendre. Une contrainte peut être donnée explicitement, en listant les combinaisons autorisées, ou implicitement, par exemple par une expression algébrique.*

Une solution à un CSP est une assignation de valeurs à toutes ses variables en répondant à toutes ses contraintes. Il peut s'agir de [Bar05] :

- Une seule solution, sans préférence,
- Toutes les solutions,
- Une solution optimale, ou du moins une bonne solution, compte tenu de certaines fonctions objectives définies en termes de toutes ou une partie des variables.

4.2.2 Techniques de résolution d'un CSP

Les problèmes de satisfaction de contraintes sont de nature combinatoire. Ainsi, un algorithme qui garantit de trouver une solution qui satisfait toutes les contraintes, en supposant qu'une telle solution existe, est énumératif. Par conséquent, le temps nécessaire pour trouver toutes les solutions croît exponentiellement avec le nombre de variables [PJ97]. L'énumération des solutions possibles se fait par la génération et l'exploration d'un arbre de recherche.
Il n'existe pas une méthode universelle capable de résoudre tous les CSP d'une manière efficace. De nombreuses techniques et méthodes ont été conçues, développées et testées pour résoudre des CSP avec plus ou moins de succès selon la nature des problèmes. Principalement, elles peuvent être des méthodes de résolution complètes ou bien incomplètes. Nous nous intéressons dans cette section à la résolution des CSP par les méthodes complètes qui sont capables de donner à la fin de la recherche une solution

réalisable, voire toutes les solutions réalisables, si elles existent. Comme ce chapitre n'est pas destiné à fournir une étude de toutes les techniques de résolution existantes, nous présentons brièvement les plus importantes (celles que nous allons utiliser par la suite). Cependant, plus de détails peuvent être trouvés dans [BPS99, Kum92, MS01, Tsa95].

La méthode Generate-and-test (GT)

La méthode *Generate-and-test* (GT) recherche systématiquement l'espace des affectations complètes. Autrement dit, elle consiste à générer toutes les combinaisons possibles de valeurs des variables. Par la suite, elle teste pour chaque affectation complète (une combinaison) si elle satisfait toutes les contraintes. Si le test échoue, c'est-à-dire qu'il existe au moins une contrainte non satisfaite, l'algorithme tente une autre affectation complète. L'algorithme s'arrête dès qu'une affectation complète satisfait toutes les contraintes définies, c'est la solution du problème ; ou bien toutes les affectations complètes sont explorées, c'est-à-dire que la solution n'existe pas.
Le nombre de combinaisons considérées par cette méthode est égal à la taille du produit cartésien de tous les domaines des variables. Ce qui n'est pas très efficace parce que la méthode génère de nombreuses affectations de valeurs erronées à des variables et qui sont rejetées lors de la phase de test (i.e. ne respectent pas les contraintes). De plus, les instanciations en conflit ne sont pas prises en compte lors de la génération d'autres affectations [Kum92].

La technique du backtraking

Un algorithme plus efficace pour effectuer une recherche systématique est le *backtraking*. Il tente progressivement d'étendre une affectation partielle qui spécifie des valeurs consistantes pour certaines variables, vers une affectation complète. Ceci consiste à choisir itérativement une valeur pour une autre variable qui soit consistante avec les valeurs de la solution partielle actuelle. Si une affectation partielle viole l'une des contraintes, le retour en arrière est effectué sur la dernière variable instanciée et qui a encore des alternatives disponibles. Chaque fois qu'une instanciation partielle viole une contrainte, le retour en arrière permet d'éliminer un sous-ensemble de l'espace de recherche (le produit cartésien de tous les domaines des variables). Par conséquent, le *backtraking* est plus efficace que la technique du *Generate-and-test* [Bar99]. Cependant, il présente certains inconvénients : la découverte redondante des inconsistances locales et la détection tardive des conflits.
Afin de remédier à ces problèmes et ainsi améliorer l'efficacité du *backtraking*, plusieurs propositions ont été appliquées [Kum92] :

- Employer les techniques de retours arrière intelligents. Elles sont des méthodes rétrospectives (*look back*) qui portent sur l'analyse et la détermination des raisons de l'échec.
- Tenter de prévenir à l'avance les inconsistances futures (principe du *look-ahead*), en combinant le *backtracking* et la propagation de contraintes (voir la section suivante).
- Définir des heuristiques sur l'ordre des variables à instancier et les valeurs dans les domaines des variables (section 4.6.1).

La consistance des contraintes

L'idée principale de la consistance est la réduction de l'espace de recherche afin de rendre moins difficile la résolution du problème. Pour ce faire, les méthodes de résolution complètes essayent de supprimer certaines valeurs dans les domaines des variables, des valeurs dites inconsistantes. Une valeur est jugée inconsistante si elle n'est pas compatible avec une ou plusieurs contraintes. La notion de consistance est donc à associer à la notion de contrainte. En effet, une contrainte force les variables à

ne prendre que certaines valeurs, la consistance intervient là où des valeurs d'un domaine ne peuvent en aucun cas satisfaire cette contrainte. La propriété de consistance pour une contrainte est atteinte lorsque plus aucune valeur ne peut être supprimée.

A chaque type de contrainte est associé un type de consistance. Ainsi, il existe principalement trois types [Kum92, Tsa95] :

— La consistance de nœud pour les contraintes unaires.
— La consistance d'arc pour les contraintes binaires.
— La consistance de chemin pour les contraintes n-aires.

La propagation de contraintes

Les techniques de recherche systématique n'ont pas prouvé leur efficacité d'être suffisamment efficaces pour résoudre un CSP [Bar05]. Différents algorithmes qui combinent un algorithme de recherche systématique avec la notion de consistance ont été introduits. Ils sont basées sur l'idée de réduire l'espace de recherche par la *propagation de contraintes*. Cette dernière représente un mécanisme d'inférence qui vise à réduire les parties de l'espace de recherche devant être visitées en éliminant celles représentant une inconsistance [BSR10]. Pour déterminer les valeurs inconsistantes, un *algorithme de filtrage* qui se base sur la notion de consistance est associé à chaque contrainte.
Avant l'intégration de la propagation de contraintes, l'algorithme de résolution se résumait à instancier les variables les unes après les autres tout en testant si elles ne violent pas les contraintes. Néanmoins, l'utilisation de la consistance (à différents niveaux) et la propagation permet de réduire l'espace de recherche et ainsi converger vers un problème plus simple à résoudre à chaque nœud et ainsi limiter les points de choix. Au niveau de chaque nœud de l'arbre de recherche, la propagation est entrelacée avec une procédure d'*énumération*. Cette dernière essaie d'affecter une valeur à chacune des variables, l'une après l'autre. Quand une valeur est choisie dans son domaine pour une variable, la propagation de contraintes est relancée pour réduire le domaine des autres variables sous l'hypothèse courante. Si une contradiction apparaît au cours du processus de résolution (violation de contrainte), la procédure revient en arrière (backtrack) pour tester d'autres valeurs. Le processus s'arrête quand toutes les contraintes sont propagées et qu'une valeur est assignée à chaque variable ce qui forme une solution. Si aucune solution n'est trouvée après combinaison de toutes les valeurs des domaines des variables, le CSP n'a pas de solution.

4.3 Le problème de construction de séquences de code vu comme un CSP

Comme expliqué précédemment dans la section 4.1, le problème de construction de séquence de code pourrait être présenté comme suit : étant donné un état mémoire de départ et un état mémoire d'arrivée, il faudrait trouver la séquence d'instructions à ajouter pour lier les deux états tout en respectant un ensemble de contraintes. Ceci revient à un problème de satisfaction des contraintes. La construction de cette séquence doit résoudre deux principaux problèmes : choisir une (ou plusieurs) instruction(s) parmi celles définies dans la spécification de la machine virtuelle [Ora15b] et calculer l'état mémoire qui la précède dans le but de converger vers l'état mémoire souhaité.

Comme indiqué auparavant, aucun problème similaire au notre n'a été discuté dans la littérature. Cependant, nous avons pu trouver une source d'inspiration dans un travail s'inscrivant dans un autre domaine et ayant un objectif totalement différent. Il s'agit des travaux de Charreteur et Gotlieb présentés dans [CG10]. Ils ont proposé une nouvelle méthode de génération automatique de données

de test, basée sur la programmation par contraintes, pour des programmes en bytecode Java. Leur méthode s'appuie sur un modèle à contraintes de la sémantique relationnelle du bytecode Java. Pour cela, la notion d'état mémoire sous contraintes (EMC) a été introduite. Un EMC contient l'information connue sur l'état de la mémoire à un instant de la résolution du problème. Chaque instruction bytecode est alors considérée comme étant une relation entre deux EMC : les états avant et après son exécution. Leur méthode de génération automatique de données de test se base sur un parcours du graphe de flot de contrôle de la méthode à tester. Il part de l'instruction à atteindre (objectif de test) pour "remonter" vers le point d'entrée de la méthode afin d'énumérer les données d'entrée permettant de couvrir une séquence d'instructions menant à l'objectif. Ceci dit, l'aspect novateur de leur approche est la définition d'un modèle de contraintes pour chaque instruction bytecode permettant de revenir en arrière lors de l'exploration du programme bytecode à tester. En faisant une projection sur notre problème, c'est précisément cette capacité qui nous intéresse pour calculer l'état mémoire précédant une instruction choisie lors de la construction de notre séquence de code.

Le formalisme CSP est assez générique et abstrait, il n'impose aucune limitation sur la nature et le nombre de contraintes, ni sur les types des domaines. La modélisation d'un problème sous forme de CSP nécessite d'identifier un ensemble de variables, un ensemble de domaines et un ensemble de contraintes. Notre problème de construction de séquence bytecode est défini comme suit :

Etant donné :

- Un ensemble fini de variables X, représentant toutes les instructions bytecode à ajouter (la séquence à trouver),
- Un domaine discret fini D, représentant l'ensemble des instructions bytecode définies dans la spécification de la machine virtuelle [Ora15b]. Chaque variable x_i dans X prend une valeur de l'ensemble D c'est-à-dire que les valeurs des variables instanciées seront sélectionnées de cet ensemble.
- Un ensemble de contraintes C représentant toutes les contraintes à satisfaire lors de la construction de la séquence (i.e. instanciation des variables de l'ensemble X)

Trouver :

Un ensemble de solutions, où une solution est une séquence d'instructions bytecode telles que chaque instruction est une instanciation d'une variable x_i de X respectant un ensemble de contraintes.

4.3.1 Les variables et leurs domaines

Nous considérons que chaque instruction à trouver est une variable à instancier. L'ensemble des variables est $X = \{x_1, x_2, ..., x_n\}$ où n est la longueur de la séquence à trouver. Une instruction bytecode est définie par son opcode et zéro ou plusieurs opérandes. Ainsi, instancier une variable x_i consiste à lui associer un opcode (l'identifiant unique d'une instruction bytecode).
Le domaine D contient tous les opcodes possibles correspondant à des instructions bytecode tels que définis dans la spécification [Ora15b]. C'est un ensemble fini de valeurs hexadécimales allant de `0x00` à `0xB8`. Donc, D ={`0x00, 0x01, 0x02, ..., 0xB7, 0xB8`}. Chacune des variables x_i peut prendre une valeur du même domaine D.

4.3.2 Les contraintes

Dans notre cas, nous proposons de définir deux catégories de contraintes : *générales* et *spécifiques*.

4.3.2.1 Les contraintes générales

Elles sont communes à toutes les instructions bytecode, i.e. elles doivent être respectées à chaque instanciation d'une variable x_i. Les contraintes sont :

- La taille de la pile des opérandes ne doit pas dépasser la valeur maximale stockée dans l'en-tête de la méthode courante (appelée *MaxStack*),
- Le nombre de variables locales ne doit pas dépasser la valeur maximale stockée dans l'en-tête de la méthode courante (appelée *MaxLocal*),
- L'instruction choisie ne doit pas causer de débordement, vers le haut ou vers le bas (i.e. *overflow*/*underflow*), dans la pile des opérandes,
- Les éléments produits/consommés par les instructions manipulant des variables locales doivent être compatibles avec la liste de variables locales,
- Les types des éléments produits par chaque instruction choisie doivent être compatibles avec l'état actuel de la pile des opérandes.

4.3.2.2 Les contraintes spécifiques

Ces contraintes sont propres à chaque instruction bytecode. Elles sont basées sur la sémantique de cette dernière telle que définie dans la spécification [Ora15b]. Cet ensemble de contraintes peut changer dynamiquement en fonction de la valeur que prend une variable. *À* chaque instruction sont associées trois contraintes. Elles concernent :

- La pré-condition, les types des éléments consommés par l'instruction.
- La post-condition, les types des éléments produits par l'instruction.
- L'utilisation d'une variable locale. Le cas échéant, un couple (`type, index`) est donné.

Nous présentons ci-dessous deux exemples d'instructions bytecode avec leurs contraintes spécifiques.

4.3.2.3 Exemple 1 : l'instruction «*sadd*»

Comme indiqué dans la figure 4.2, l'instruction `sadd` (son opcode est `0x41`) a zéro opérandes, elle dépile (consomme) deux valeurs de type `short` du sommet de la pile d'opérandes (c'est sa pré-condition), fait leur somme puis empile (produit) le résultat qui est aussi une valeur de type `short` (c'est sa post-condition). Elle n'utilise pas de variables locales.

Les contraintes associées à cette instruction sont donc :

- $Pre\_condition = \{short, short\}$
- $Post\_condition = \{short\}$
- $Var\_locales = \phi$

4.3.2.4 Exemple 2 : l'instruction «*aload*»

Comme indiqué dans la figure 4.3, l'instruction `aload` (son opcode est `0x15`) a une opérande (c'est un index dans la liste des variables locales qui doit être de type `objectRef`), elle ne consomme aucun élément de la pile des opérandes (pré-condition vide). Elle charge une valeur de type `objectRef` de la liste des variables locales (à l'index donné comme paramètre de l'instruction) vers la pile des opérandes (c'est sa post-condition).

Les contraintes associées à cette instruction sont donc :

- $Pre\_condition = \phi$
- $Post\_condition = \{objectRef\}$
- $Var\_locales = \{(objectRef, index)\}$

sadd
Add short

Format
sadd

Forms
sadd = 65 (0x41)

Stack
..., value1, value2 –>
... , result

Description
Both value1 and value2 must be of type short. The values are popped from the operand stack. The short result is value1 + value2. The result is pushed onto the operand stack.
If a sadd instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide twos-complement format. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Fig 4.2 – Spécification de l'instruction «*sadd*» [Ora15b]

4.4 Eléments de modélisation

Partant de la formulation de notre problème comme un CSP (section 4.3), nous visons à lui trouver une solution, i.e. instancier toutes ses variables en respectant toutes les contraintes définies. La recherche d'une solution à un CSP peut être vue comme un parcours d'arbre. Un aspect important de la recherche considérée ici est que l'arbre à parcourir n'est pas donné à l'avance : il est généré à la volée.

4.4.1 Structure générale de l'arbre de recherche

Étant donné un état mémoire de départ, un état mémoire d'arrivée et la liste de toutes les instructions bytecode (parmi lesquelles nous choisirons les instructions à rajouter), nous pouvons construire un arbre de recherche où :

— La *racine* représente l'état mémoire de départ.
— Chaque *niveau* contient les instructions candidates (c'est-à-dire celles respectant les contraintes) qui appartiennent potentiellement à la solution finale (la séquence à trouver) dans le sens où elles peuvent précéder l'instruction du niveau supérieur de l'arbre.
— Chaque *nœud intermédiaire* représente un état mémoire correspondant à une instruction candidate, i.e. une instruction précédant le nœud parent (nous rappelons que nous raisonnons dans le sens opposé que celui de l'exécution).
— Chaque *feuille* représente un état final de la mémoire (éventuellement l'état d'arrivée souhaité).

La structure générale de notre arbre de recherche est donnée dans la figure 4.4. Chaque nœud *EM* est une abstraction d'un état mémoire.

aload
Load reference from local variable

Format
aload
index

Forms
aload = 21 (0x15)

Stack
... –>
... , objectRef

Description
The *index* is an unsigned byte that must be a valid index into the local variables of the current frame. The local variable at *index* must contain a reference. The objectref in the local variable at *index* is pushed onto the operand stack.

Fig 4.3 – Spécification de l'instruction «*aload*» [Ora15b]

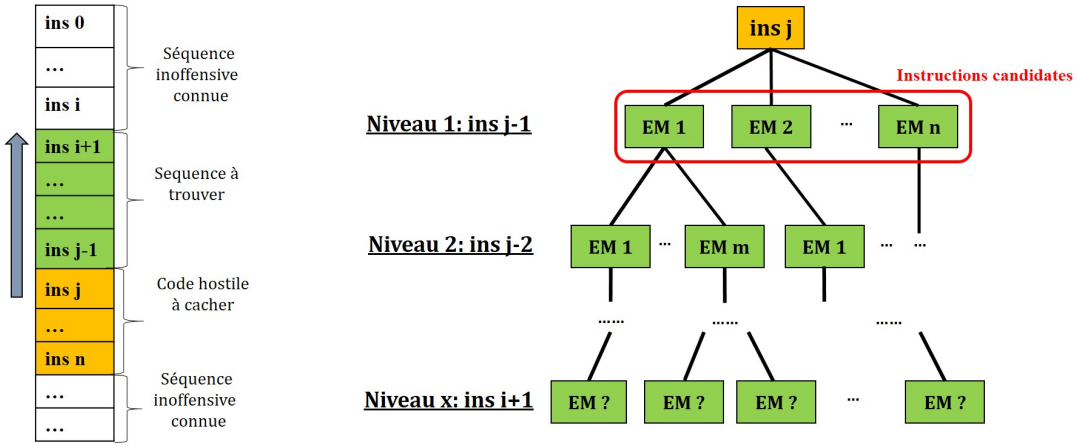

Fig 4.4 – Structure générale de l'arbre de recherche

4.4.2 Structure d'un nœud de l'arbre

Chaque nœud de l'arbre de recherche représente un état mémoire. La structure d'un nœud (figure 4.5) est composée de deux parties :

1. *Partie données* : permet de gérer les contraintes lors de la génération de l'arbre de recherche, elle comprend :
   — L'état courant de la pile des opérandes (de taille *maxStack*)
   — L'état courant de la liste des variables locales (de taille *maxLocal*)
   — Le pointeur de pile indiquant le sommet de la pile (nommé *SP* pour *Stack Pointer*)
   — L'opcode de l'instruction candidate (pour laquelle le nœud sera créé).
2. *Partie pointeurs* : permet d'assurer le parcours de l'arbre de recherche dans les deux sens (descendant et ascendant), elle englobe :
   — Un pointeur vers le nœud parent
   — Une liste de pointeurs vers les éventuels nœuds fils (chacun correspond à une instruction candidate)

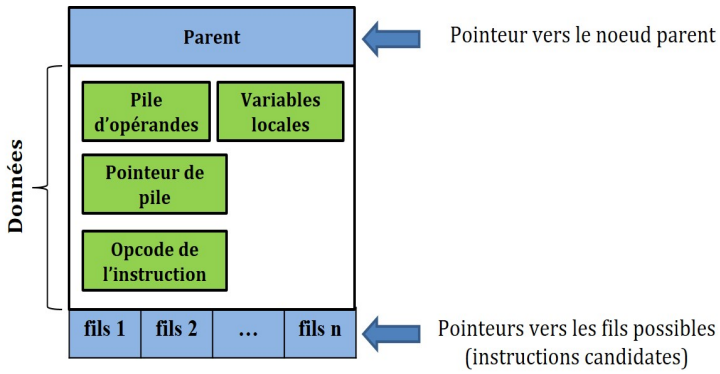

Fig 4.5 – Structure d'un nœud de l'arbre de recherche

4.5 L'approche proposée pour la génération des séquences de code

Dans cette section, nous présentons d'abord notre algorithme de parcours d'arbre (section 4.5.1), nous détaillerons ensuite comment calculer un état mémoire (section 4.5.2). Après cela, l'approche proposée pour la construction de séquence de code est illustrée à travers un exemple (section 4.5.3). Enfin, nous présentons l'étape qui suit le processus de génération des solutions à savoir la manipulation et la vérification du fichier CAP (section 4.5.4).

4.5.1 Algorithme de construction de séquences de code par parcours d'arbre

Comme expliqué dans la section 4.2.2, il existe de nombreuses techniques pour résoudre un CSP. L'algorithme 1 représente notre approche de génération/exploration d'un arbre de recherche dans le but de générer des séquences de bytecode liant deux états mémoire. Il est basé sur les éléments suivants :

— *Méthode de résolution* : backtraking
— *Stratégie du parcours d'arbre* : parcours en profondeur d'abord (*depth-first-strategy*) car l'objectif est de converger rapidement vers une solution i.e. un chemin complet de la racine vers une feuille
— *Niveau de propagation des contraintes* : comme les contraintes sont unaires, il faudrait assurer la consistance de nœud.

Les deux critères d'arrêt de l'algorithme sont :

— Atteindre l'état final souhaité dans l'arbre.
— Atteindre la profondeur maximale (sa valeur est fixée à l'avance).

L'objectif de notre approche est donc, à partir d'un état mémoire de départ, d'insérer des instructions et de recalculer l'état mémoire la précédant afin de converger vers un état mémoire d'arrivée que nous désirons rejoindre. Chaque chemin de la racine à une feuille de l'arbre (dans le sens ascendant de l'exécution) représente une solution possible, i.e. une séquence d'instructions bytecode qui peut être rajoutée au code initial. Ainsi, la construction de cette séquence doit résoudre deux principaux problèmes :

— Le choix des instructions à rajouter tout en respectant les contraintes définies (ceci est assuré par la propagation des contraintes avant chaque choix)
— Le calcul des états mémoire correspondants (un état mémoire par instruction choisie).

La fonction de décision du choix des instructions doit être accompagnée d'un mécanisme de retour en arrière (*backtraking*) si un critère d'arrêt est rencontré.

Algorithme 1 : Construction de séquence de code par parcours d'arbre

Entrées :
- Un état mémoire de départ
- Un état mémoire d'arrivée
- Un ensemble d'instructions bytecode

Sorties : Les séquences de code (suite d'instructions) reliant les deux états mémoire

```
1  Début
2      - Partant de la racine de l'arbre, trouver toutes les instructions candidates (celles qui
         peuvent précéder la racine) tout en respectant les contraintes;
3      Tant que (la profondeur maximale n'est pas atteinte) et (l'état désiré n'est pas atteint)
         faire
4          si le nœud courant n'est pas la racine alors
5              - Trouver toutes les instructions candidates du noœud courant ;
6          fin
7          - Générer tous les fils du nœud courant qui correspondent aux instructions candidates
             trouvées (un nœud par instruction candidate);
8          - Sélectionner un nœud fils, non visité, à explorer ;
9          - Calculer l'état mémoire du nœud sélectionné et le marquer comme visité;
10     fintq
11     si la profondeur maximale est atteinte alors
12         ** si l'état mémoire désiré est atteint alors
13             - Mémoriser la solution (le chemin de la racine vers ce nœud);
14             * si un nœud parent possède encore un fils non visité alors
15                 - Faire un retour arrière (backtracking) à ce nœud et le considérer comme nœud
                     courant;
16                 - Reprendre à la ligne 3;
17             sinon
18                 - Terminer la recherche (la racine ne possède plus de fils à explorer);
19             fin
20         sinon
21             - Aller à *
22         fin
23     sinon si l'état désiré est atteint alors
24         - Aller à **
25     fin
26 Fin
```

Partant de la racine de l'arbre, nous générons tous ses fils possibles (un nœud fils par instructions candidate). Ensuite, choisir le premier fils de la liste comme le nouveau nœud courant et partir dans une génération/exploration en profondeur de l'arbre de recherche. Ce processus est répété tant qu'un état désiré n'est pas atteint et que la profondeur maximale de l'arbre n'est pas atteinte aussi. Si un critère d'arrêt est rencontré, nous vérifions si l'état atteint est celui voulu. Le cas échéant, le chemin de la racine à ce nœud est mémorisé comme étant une solution possible. Ensuite, il y a retour arrière au nœud parent du nœud courant. Après, le prochain fils non visité de ce nœud parent est sélectionné comme nouveau nœud courant. Par la suite repartir dans la génération/exploration de l'arbre. Ce processus continue jusqu'à ce que le contrôle retourne au nœud racine et que tous ses fils soient visités. Tous les chemins sauvegardés lors de du processus génération/exploration constituent les solutions possibles au problème de construction de code.

Bien que théoriquement une seule solution soit suffisante pour valider l'approche, nous devons

générer plusieurs solutions. Ceci est dû au fait que les solutions trouvées doivent subir une phase de vérification afin de déterminer celles qui sont valables au sens qu'elles puissent être acceptées par le vérifieur de bytecode (voir section 4.5.4). Donc en générant plusieurs solutions, nous augmentons la chance de trouver une bonne solution.

4.5.2 Le calcul d'un état mémoire

Comme expliqué dans l'algorithme 3, le choix d'un nœud fils à explorer est accompagné du calcul de l'état mémoire correspondant. Ce calcul comprend :

— Le calcul de l'état de la pile d'opérandes
— La mise à jour du pointeur de pile
— La mise à jour de la liste des variables locales

Comme expliqué précédemment, nous raisonnons dans le sens inverse que celui de l'exécution. Ainsi, pour chaque instruction candidate (nœud fils), au lieu de dépiler sa consommation de la pile et empiler sa production sur la pile, il faudrait dépiler les éléments produits (post-condition) et empiler les éléments à consommer (pré-condition). Ainsi, à chaque création de nœud, l'état de la pile peut être obtenu comme suit :

`(Etat_pile_fils)` = `(Etat_pile_parent)` - `(Post-condition de l'instruction candidate)` + `(Pre-condition de l'instruction candidate)`

L'exemple présenté ci-dessous, illustre le calcul de l'état de la pile des opérandes en résumant les étapes d'une exécution normale et celles d'une exécution inversée (qui correspond à notre cas) pour la même instruction bytecode. Considérons :

— Une état initial de la pile d'opérandes = $\{short, objectRef, short, short\}$
— Une instruction bytecode à exécuter= *sadd*

La figure 4.6 présente une exécution normale de l'instruction *sadd* (dont la spécification a été donnée dans la figure 4.2). L'effet de cette exécution résulte au dépilement de deux valeurs de type *short* (sa pré-condition) puis à l'empilement d'une valeur de type *short* (sa post-condition).

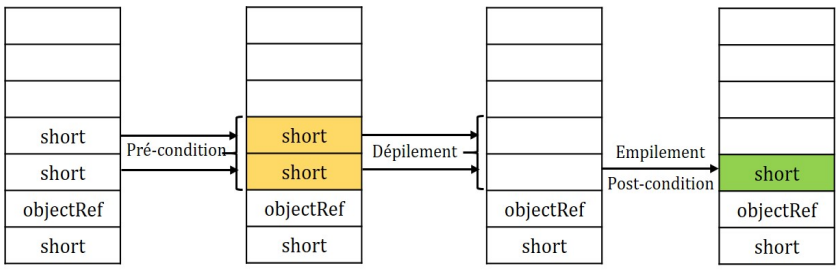

Fig 4.6 – Exécution normale d'une instruction *sadd*

La figure 4.7 présente une exécution inversée de l'instruction *sadd*. L'effet de cette exécution résulte au dépilement d'une valeur de type *short* (sa post-condition) puis à l'empilement de deux valeurs de type *short* (sa pré-condition).

4.5.3 Exemple de déroulement de l'algorithme

En guise d'illustration, nous présentons un exemple d'exécution de l'algorithme 3 (figure 4.8). Afin de simplifier la représentation de l'arbre de recherche généré, nous avons choisi de limiter la représentation d'un état mémoire à son état de pile d'opérandes et l'instruction bytecode correspondante.

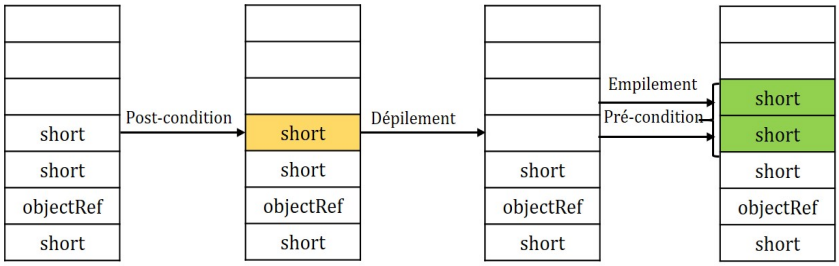

Fig 4.7 – Exécution inversée d'une instruction *sadd*

Pour avoir un arbre de recherche de taille raisonnable à présenter dans cette section, nous avons sélectionné un sous-ensemble d'instructions bytecode à considérer lors de la génération/exploration de l'arborescence. Le tableau 4.1 résume les instructions choisies ainsi que leurs contraintes spécifiques (pré-condition, post-condition et utilisation de variables locales).

Instruction	**Pré-condition**	**Post-condition**	**Variables locales**
`aload_0`	none	ObjectRef	(ObjectRef, 0)
`aload_1`	none	ObjectRef	(ObjectRef, 1)
`bspush`	none	short	none
`getfield_a_this`	none	ObjectRef	none
`sadd`	short short	short	none
`sconst_2`	none	short	none
`sload_0`	none	short	(short, 0)
`sload_1`	none	short	(short, 1)

Tab 4.1 – pré/post-conditions d'un sous-ensemble d'instructions bytecode

Considérons les entrées suivantes :

— Instruction initiale = `sload_1`
— Etat initial de la pile d'opérandes = $\{short, objectRef, short\}$
— Etat final de la pile d'opérandes =ϕ
— Liste des variables locales = $\{objectRef, short\}$
— MaxStack = 4
— MaxLocal = 4
— Profondeur maximale = 3

Partant de l'instruction `sload_1` (nœud racine), la liste des instructions bytecode est parcourue (tableau 4.1) pour sélectionner les instructions candidates compatibles avec l'état actuel de la mémoire. En d'autres termes, nous recherchons les instructions dont les post-conditions correspondent au sommet de la pile d'opérandes (un ou plusieurs éléments) et l'état des variables locales du nœud courant. Avant cela, toutes les contraintes générales doivent être respectées (section 4.3.2). Ainsi, l'instruction `sload_1` peut être précédée par `bspush`, `sadd`, `sconst_2` ou `sload_1` (`sload_0` est incompatible car la variable locale à l'index 0 est une référence et non pas un short).
Rappelons que nous effectuons une recherche en profondeur lors de la génération/exploration de notre arbre. Comme aucune priorité n'est établie entre les fils, i.e. les instructions candidates, le premier nœud fils devient le nœud courant et le processus est répété. Ainsi, l'instruction `bspush` (le premier fils de `sload_1`) peut être précédée par `aload_0` ou `getfield_a_this`. Après cela, nous sélectionnons les instructions candidates pour `aload_0` qui sont `bspush`, `sadd`, `sconst_2` ou `sload_1`. Dans l'étape suivante, lors de la sélection de l'instruction `bspush`, nous constatons que la profondeur maximale

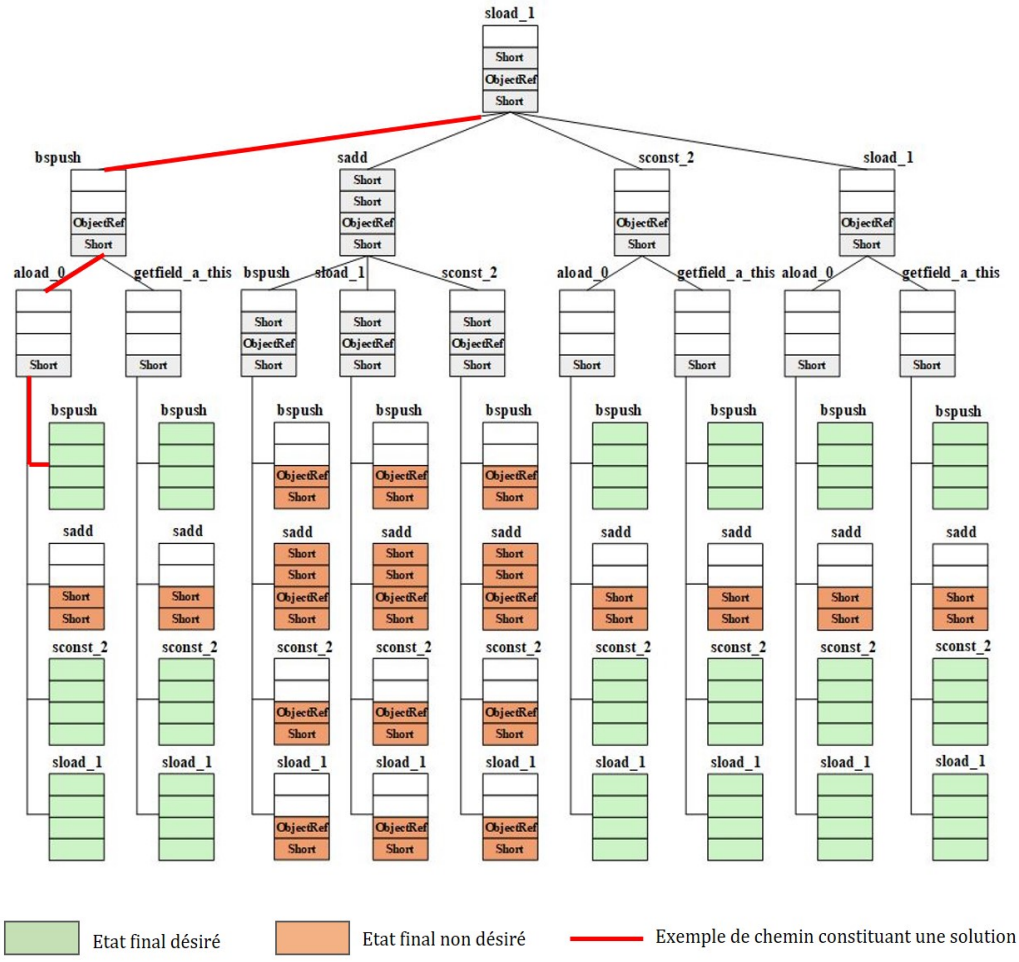

Fig 4.8 – Exemple de déroulement de l'algorithme 1

est atteinte et que l'état actuel de la pile d'opérandes correspond à l'état désiré (pile vide). Par conséquent, le chemin de la racine à cette instruction (`sload_1, bspush, aload_0, bspush`) est sauvegardé comme étant une solution possible. Par la suite, nous revenons au nœud parent (`aload_0`) pour explorer le prochain fils non visité (`sadd`). La profondeur maximale est atteinte et l'état de la pile d'opérandes n'est pas celui attendu. Donc, nous revenons en arrière au nœud parent et répétons le même processus jusqu'à ce que le nœud racine n'aie plus de fils à visiter.
Ainsi au final pour l'exemple traité, 18 chemins sur 36 sont considérés comme étant potentiellement des solutions à notre CSP.

4.5.4 Vérification des solutions générées : manipulation de fichier CAP

La liste des solutions générées à l'étape précédente doit être analysée pour détecter lesquelles sont valables, dans le sens où elles pourraient être générées par un compilateur. Pour celà, nous procédons en deux étapes comme présenté dans la figure 4.9.

4.5.4.1 La première étape : manipulation de fichier CAP

Suite à la génération de l'arbre de recherche, chaque solution sauvegardée, i.e. une séquence d'instructions bytecode, doit être insérée dans un fichier CAP correct (en utilisant l'outil *Cap manipulator*[1]).

1. Disponible sur : https ://bitbucket.org/ssd/capmap-free

Par la suite, ne garder que celles qui pourraient être acceptées par le vérificateur de bytecode. Comme chaque nouveau fichier CAP est le fichier CAP original auquel nous avons rajouté la séquence trouvée (i.e. une solution), nous aurons autant de fichiers CAP produits que de solutions générées. Dans cette étape, deux principaux problèmes doivent être résolus :

— Adapter le format de la solution à celui d'un fichier CAP tel que défini dans la spécification [Ora15b] (i.e. extraire les informations des solutions générées et les transformer dans un format exploitable)

— Compléter les instructions par les opérandes manquantes en leur associant des valeurs correctes. Certaines valeurs doivent être calculées (offset d'un saut, par exemple). Tandis que d'autres doivent être récupérées à partir du fichier CAP à modifier (argument référencé dans le composant Constant_Pool, par exemple). Ceci nécessite une maîtrise des composants du fichier CAP, essentiellement les composants : Constant_Pool et Method.

4.5.4.2 La deuxième étape : vérification de fichier CAP

Chaque fichier CAP créé dans la première étape (4.5.4.1), est converti en un fichier *Class*, puis en un fichier *Java*. Avec ce dernier, nous reproduisons le processus de compilation en fichier *Class* puis conversion en fichier CAP. À la fin, le fichier CAP original (créé à la première étape) et celui qui vient d'être créé sont comparés. Si les deux fichiers sont identiques, nous pouvons conclure que le programme correspondant pourrait être généré par un compilateur. Donc, la solution en question est acceptée. Dans l'autre cas (les deux fichiers sont différents), la solution est rejetée.

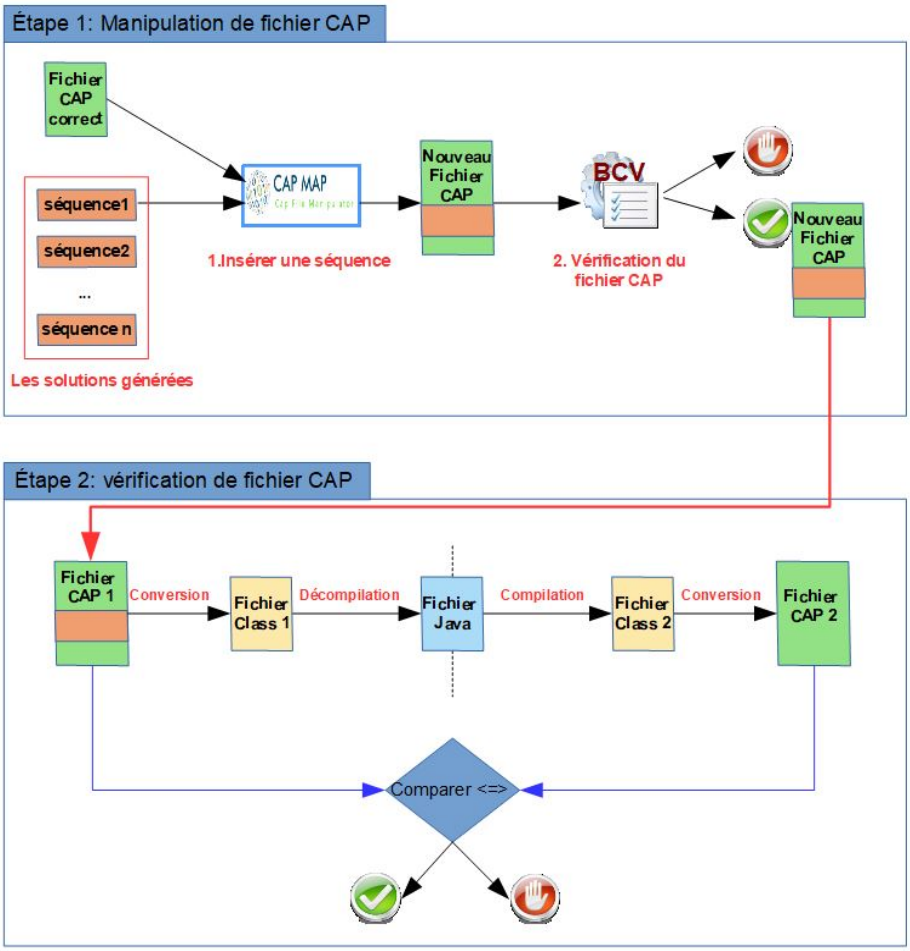

Fig 4.9 – Processus de manipulation et vérification de fichier CAP

4.6 Optimisation de l'approche : utilisation des heuristiques

4.6.1 Stratégies d'énumération : heuristiques de prise de décision

De nombreuses recherches [BR96] ont montré que l'ordre dans lequel sont considérées les variables pour leur affecter une valeur, ainsi que l'ordre dans lequel sont affectées les valeurs d'un domaine à une variable, peuvent influencer le temps de résolution. En effet, les heuristiques d'ordonnancement des variables/valeurs peuvent jouer un rôle très important dans l'amélioration de l'efficacité de l'algorithme de recherche (temps de résolution) en influant sur la taille de l'espace de recherche [Kum92]. Différentes stratégies d'énumération (ou heuristiques d'ordonnancement) ont été proposées dans le but d'obtenir une solution plus rapidement. Ces heuristiques peuvent être statiques, l'ordre est fixé au début et reste inchangé tout au long de la recherche, ou bien dynamiques, l'ordre peut varier dynamiquement au cours de la recherche. De plus, ces heuristiques peuvent être des stratégies spécifiques au problème à traiter, ou bien des stratégies génériques applicables sur tous les problèmes indépendamment de leur nature. Parmi ces dernières, nous pouvons citer [Tsa95, BR96] :

- La cardinalité maximale : la prochaine variable à instancier doit avoir le plus grand nombre de contraintes avec les variables déjà instanciées.
- Le degré maximum : la prochaine variable à instancier doit avoir le plus grand nombre de contraintes.
- dom ou MRV : dom ou MRV pour *Minimum Remaining Value* consiste à choisir la variable possédant le plus petit domaine.
- dom/deg : La prochaine variable à instancier est la variable qui minimise le rapport taille du domaine/nombre de contraintes.

4.6.2 Nos heuristiques

Dans l'approche proposée (section 4.5), les nœuds fils (instructions candidates) sont explorés sans préférences, c'est-à-dire avec la même chance, ce qui n'est pas vraiment efficace dans la pratique. En d'autres termes, les combinaisons qui ne correspondent pas à des programmes réels seront explorées du moment qu'elles respectent les contraintes. Un moyen d'optimiser le processus de recherche consiste à introduire des heuristiques pour une convergence plus rapide vers des solutions plus réalistes. Pour cela, nous avons défini une stratégie d'énumération spécifique à notre problème. L'idée est de pondérer les instructions candidates de chaque nœud de manière qu'elles soient ordonnées et explorées en fonction de leurs priorités : le meilleur nœud d'abord. Cet ordre d'exploration des nœuds fils change « dynamiquement » en fonction du nœud courant en question.
Afin d'appliquer cette stratégie d'énumération, un travail préliminaire consiste en la génération de fichiers statistiques pour chaque instruction bytecode. Ceci est fait en se basant sur la fréquence de transition d'une instruction vers les instructions précédentes. Les données sont enregistrées à partir d'un ensemble d'applets Java Card (plus précisément des fichiers *JCA* [2]). Ainsi, lorsqu'une étape de génération est lancée, les statistiques correspondant à l'instruction en cours sont chargées. Les statistiques incluent les noms des instructions triés par fréquence de transition (la première instruction étant la plus fréquente). Selon la nature du nœud considéré, nous avons développé deux types de statistiques :

- Les bi-grams, statistiques appliquées uniquement au nœud racine de l'arbre de recherche
- Les tri-grams, statistiques appliquées aux nœuds intermédiaires de l'arbre de recherche

2. Un fichier JCA (Java Card Assembly) est une représentation textuelle du contenu d'un fichier CAP

4.6.2.1 Utilisation des bi-grams

Un script *Python* est développé pour analyser des fichiers *JCA* et calculer les fréquences de transition pour chaque instruction bytecode. Initialement, le script génère une matrice de transition entre toutes les instructions, elle correspond à la fréquence de chaque instruction précédente. Avec cette matrice, le script produit pour chaque instruction une liste triée des instructions pouvant la précéder (de la plus fréquente vers la moins fréquente). Un exemple des statistiques bi-grams générées pour certaines instructions est donné dans le tableau 4.2.

Instruction	**Instructions précédentes possibles**
`aaload`	`sload_1 sload sload_3 sload_2 ssub getfield_b_this sconst_0 ...`
`aconst_null`	`aload_0 areturn ifnull new putfield_a putstatic_a`
`astore`	`goto checkcast aaload newarray aload_1 getfield_a_this aload_3 ...`
`bspush`	`bspush dup sconst_0 aload_1 aload_0 sload baload aload_2 ...`
`if_acmpeq`	`aaload aload`
`sadd`	`sconst_1 sload sconst_2 bspush sadd sconst_4 sload_2 sload_3 ...`

Tab 4.2 – Exemple des statistiques bi-grams

Ainsi pour chaque instruction bytecode, le fichier des statistiques fournit une liste différente des instructions précédentes possibles. De ce fait, le nombre de ces instructions candidates est différent d'une instruction à une autre. Cela permet de réduire le nombre de branches possibles pour un nœud donné et permet ainsi une convergence plus rapide vers des solutions plus réalistes.

4.6.2.2 Utilisation des tri-grams

Un second script *Python* est développé pour une autre exploitation des fichiers JCA. Il permet de générer les fichiers des statistiques tri-grams afin de diriger la sélection des fils d'un nœud en se basant sur la connaissance de son nœud parent. Une table des instructions candidates (i.e. fils possibles) est générée pour chaque instruction. Chaque table est stockée dans un fichier indépendant. La figure 4.10 résume le principe des tri-grams. Supposons que l'instruction en cours est *ins2*. Nous essayons de trouver l'instruction *ins1* (toutes les possibilités des nœuds fils) de telle sorte que *ins2* soit précédée par *ins1* et suivie de *ins3* (nœud parent). Chaque fichier des statistiques tri-grams est organisé comme suit :

— Le nom du fichier est le nom de l'instruction en cours.
— La première colonne est l'instruction suivante (le nœud parent dans notre arbre de recherche).
— Le reste de la ligne est constitué des instructions précédentes possibles (nœuds fils, i.e. les instructions candidates) triées dans l'ordre décroissant de leurs priorités.

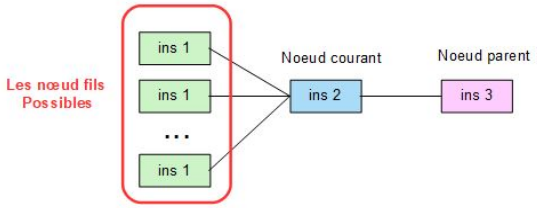

Fig 4.10 – Le principe des tri-grams

Ainsi, lorsque la génération des fils d'un nœud est lancée, le programme ouvre le fichier des statistiques tri-grams qui correspond au nom de l'instruction en cours. Il recherche la ligne qui

correspond au nom de son nœud parent dans la première colonne, et charge le reste de la ligne comme étant les instructions candidates à explorer (classées par ordre décroissant de priorité).

sload	
Instruction suivante	**Instructions précédentes possibles**
`aaload`	`getfield_a_this`
`aload`	`aload sload sadd sstore`
`baload`	`getfield_a_this aload_2 aload_1 aload_3 aload aload_0`
`dup`	`aload_2 aload_3 getfield_a_this aload`
`sload`	`getfield_a_this sstore sinc aload_2 aload_0 astore sload`

Tab 4.3 – Exemple des statistique tri-grams pour l'instruction *sload*

Par exemple, comme indiqué dans le tableau 4.3, l'instruction `sload` suivie d'un `aload` (nœud parent) pourrait être précédée de (nœuds fils possibles) : `aload`, `sload`, `sadd`, `sstore` dans l'ordre décroissant de priorité.

4.6.3 Version optimisée de l'algorithme de construction de séquences de code par parcours d'arbre

En appliquant les heuristiques introduites plus haut (section 4.6.2), nous obtenons la nouvelle version de l'algorithme de construction de séquence de code (algorithm 2). Les changements apportés sont mis en évidence dans une autre couleur de texte (en rouge).

Donc pour résumer, le choix des instructions candidates et l'ordre d'exploration des nœuds fils correspondants sont basés sur :

— Les bi-grams pour le nœud racine ;
— Les tri-grams pour les nœuds intermédiaires. Pour chacun d'entre eux, il y a chargement du fichier des statistiques tri-grams correspondant.

Algorithme 2 : Construction de séquence de code par parcours d'arbre avec utilisation des heuristiques

Entrées :
- Un état mémoire de départ
- Un état mémoire d'arrivée
- Un ensemble d'instructions bytecode
- les fichiers statistiques (bi-grams et tri-grams)

Sorties : Une séquence de code (suite d'instructions) reliant les deux états mémoire

```
1  Début
2      - Charger le fichier des statistiques bi-grams ;
3      - Repérer dans ce fichier la ligne correspondant au nœud racine ;
4      - Parmi les instructions de la ligne repérée, trouver toutes les instructions candidates (celles
         qui peuvent précéder la racine tout en respectant les contraintes);
5      Tant que (la profondeur maximale n'est pas atteinte) et (l'état désiré n'est pas atteint)
         faire
6          si le nœud courant n'est pas la racine alors
7              - Charger le fichier des statistiques tri-grams du nœud courant ;
8              - Repérer dans ce fichier la ligne correspondant à son nœud parent ;
9              - Parmi les instructions de la ligne repérée, trouver toutes les instructions candidates
                 ;
10         fin
11         - Générer tous les fils du nœud courant qui correspondent aux instructions candidates
             trouvées (un nœud par instruction candidate);
12         - Sélectionner le nœud fils le plus fréquent, non visité, à explorer ;
13         - Calculer l'état mémoire du nœud sélectionné et le marquer comme visité;
14     fintq

15     si la profondeur maximale est atteinte alors
16         ** si l'état mémoire désiré est atteint alors
17             - Mémoriser la solution (le chemin de la racine vers ce nœud);
18             * si un nœud parent possède encore un fils non visité alors
19                 - Faire un retour arrière (backtracking) à ce nœud et le considérer comme nœud
                     courant;
20                 - Reprendre à la ligne 5;
21             sinon
22                 - Terminer la recherche (la racine ne possède plus de fils à explorer);
23             fin
24         sinon
25             - Aller à *
26         fin
27     sinon si l'état désiré est atteint alors
28         - Aller à **
29     fin
30 Fin
```

4.6.4 Exemple de déroulement de l'algorithme utilisant des heuristiques

Afin d'illustrer l'amélioration apportée par la nouvelle version de l'algorithme de construction de séquence de code, nous préférons reprendre l'exemple présenté dans la section 4.5.3. Et ainsi voir les changements effectués sur l'arbre de recherche résultant du déroulement de l'algorithme 2 (figure 4.11). Rappelons que le jeu des instructions bytecode considéré dans l'exemple est constitué des instructions suivantes : `aload_0, aload_1, bspush, getfield_a_this, sadd, sconst_2, sload_0, sload_1`.

Partant de l'instruction `sload_1` (nœud racine), le fichier des bi-grams est chargé, la ligne cor-

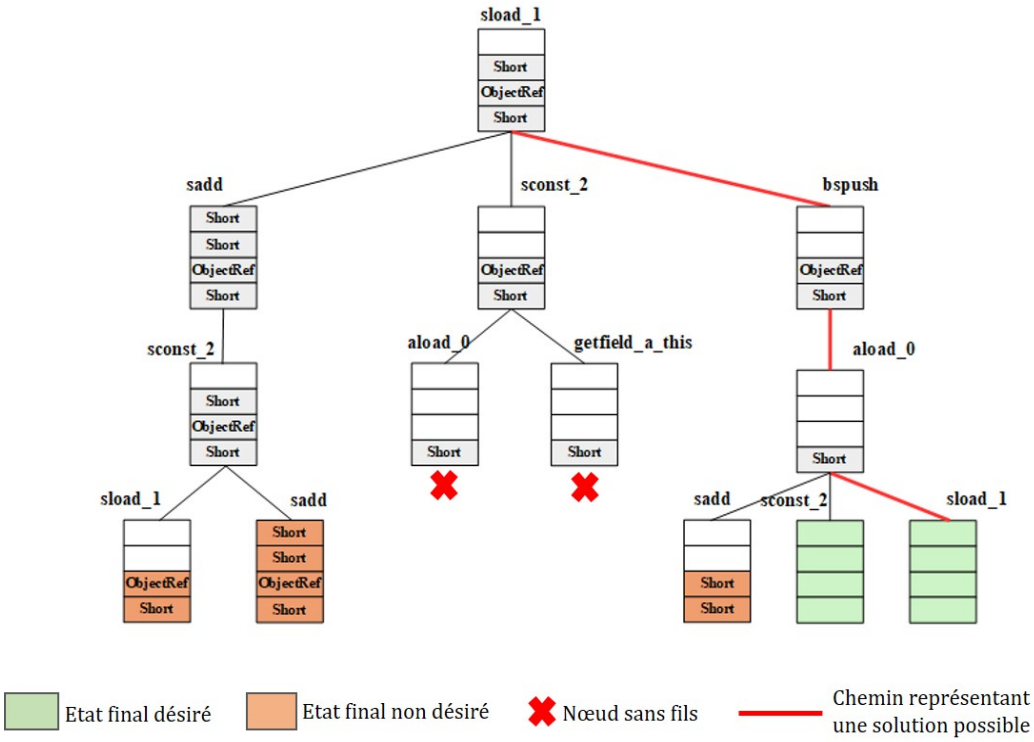

Fig 4.11 – Exemple de déroulement de l'algorithme 2

respondant à cette instruction est repérée (tableau 4.4) pour sélectionner les instructions candidates compatibles avec l'état actuel de la mémoire. Ainsi, l'instruction `sload_1` peut être précédée par les instructions `sadd`, `sconst_2` ou `bspush` dans l'ordre décroissant de leur priorité.
Par la suite, en appliquant notre heuristique, l'instruction `sadd` devient le nœud courant à explorer. Pour cela, le fichier des statistiques tri-grams correspondant est chargé, la ligne relative au nœud parent (`sload_1`) est repérée pour sélectionner les instructions candidates pouvant précéder l'instruction `sadd`. Dans ce cas, il s'agit uniquement de l'instruction `sconst_2` qui devient à son tour le nœud courant à explorer. Le processus de génération/exploration, accompagné du *backtracking* et le chargement des fichiers des statistiques tri-grams est répété jusqu'à ce que le nœud racine n'aie plus de fils à visiter.

Les statistiques tri-grams utilisées dans cet exemple, pour la génération/exploration de l'arbre de recherche, sont données dans le tableau 4.5.

Instruction	**Instructions précédentes possibles**
`sload_1`	getfield_a_this, aload_0, sstore_1, sinc, goto, getfield_s, getfield_s_this, if_scmpeq, sconst_0, dup, getfield_b_this, if_scmplt, sadd, sconst_1, sconst_2, sload_2, sreturn, aload_2, bspush, getstatic_a, if_scmpne, ifeq, pop, sload, sstore, sstore_2, sstore_3, aload, astore_2, sand, sload_0, sload_3, baload, bastore, if_scmpgt, if_scmple, ifne, putfield_s, return

Tab 4.4 – Statistiques bi-grams de l'instruction sload_1 (nœud racine)

Comme nous pouvons clairement le constater dans la figure 4.6.4, la taille de l'arbre de recherche résultant de l'application de l'algorithme 2 a considérablement diminuée. Ainsi au final pour l'exemple traité, 2 chemins sur 7 (au lieu de 18 sur 36) sont considérés comme étant potentiellement des solutions à notre CSP.

Instruction	**Instruction suivante (parent)**	**Instructions précédentes possibles (fils)**
sadd	sload_1	sconst_2, sload
sconst_2	sload_1	aload_0, getfield_a_this, sload
bspush	sload_1	aload_0 aload_2
sconst_2	sadd	sload, sload_2, baload, sload_3, sadd, sload_1, getfield_s_this, saload, sdiv, smul
aload_0	sconst_2	putfield_a, putfield_b, putfield_s, ifeq, ifne, ifnonnull
getfield_a_this	sconst_2	getfield_a_this, pop, bastore, aload, putfield_a, sconst_5, sstore_2
aload_0	bspush	putfield_a, sadd, if_scmplt, putfield_s, ifeq, putfield_b, sstore, astore_3, goto, if_scmpge, ifgt, return, sconst_2, sload_1, sload_3, ssub

Tab 4.5 – Statistique tri-grams

4.7 Conclusion

Dans ce chapitre, nous avons montré que le problème de construction de séquences de code se ramène à un problème de résolution de contraintes. En se basant sur des fondements théoriques du domaine des CSPs, nous avons proposé une modélisation du problème qui a servi par la suite à la mise au point d'une solution adéquate (parcours d'arbre pour la génération des séquences de code). De plus, une optimisation de cette approche a été proposée pour améliorer les résultats obtenus et y converger plus rapidement. Dans le chapitre suivant, nous nous intéressons au deuxième sous-problème soulevé au début de ce travail : la désynchronisation de code. Nous montrerons comment l'approche proposée dans ce chapitre et celle qui sera présentée dans le suivant sont complémentaires afin de répondre au problème principal traité sans cette thèse.

CHAPITRE 5

Désynchronisation de code

Sommaire

Dans une carte à puce, la défaillance du système peut être causée par des évènements indésirables qui touchent aux quatre principales propriétés de sécurité : l'intégrité du code ou des données et la confidentialité du code ou des données. L'intégrité du code est la propriété la plus sensible car elle permet à un attaquant de modifier le code qui s'exécute ce qui conduit par conséquence à toucher aux trois autres propriétés i.e. l'intégrité des données et la confidentialité du code ou des données [Bou14]. Le mécanisme que nous allons présenter dans ce chapitre s'inscrit dans la catégorie des évènements indésirables touchant l'intégrité du code. Il s'agit de ce que nous avons appelé *la désynchronisation de code*, visant à dissimuler un code hostile dans un autre code inoffensif qui pourrait être chargé dans la carte sans qu'il ne soit détecté par les mécanismes de sécurité embarqués ou éventuellement une analyse du code. Ceci dans le but de l'activer par la suite à travers une injection de faute. Dans ce chapitre, nous présenterons le principe général de ce mécanisme (section 5.1) qui sera illustré à travers un exemple de preuve de concept (section 5.3). Par la suite, nous nous fixerons sur un modèle de désynchronisation spécifique (section 5.2) qui sera traité en détails dans les sections suivantes (sections 5.4 et 5.5).

5.1 Comment dissimuler un code hostile : principe de la « désynchronisation »

L'objectif principal étant de dissimuler un code malveillant, ce dernier doit subir des changements qui transforment sa structure ainsi que sa sémantique afin qu'il apparaisse sous une forme différente de l'originale (inoffensive et correcte aussi). Cependant, une fois dans la carte, le code hostile de départ doit pouvoir être retrouvé afin d'activer son comportement malveillant et ceci à travers une attaque par injection de faute. Donc pour résumer, le code à construire possède deux sémantiques correctes vis-à-vis de la spécification de la JCVM : une première sémantique inoffensive avant l'injection de la faute i.e. le code construit (code caché) et une seconde sémantique représentant un comportement malveillant après l'injection de la faute (code hostile retrouvé).

Dans le chapitre précédent, nous avons montré que ce problème de dissimulation de code revient en partie à un problème de construction de séquence de code. Autrement dit, pour cacher un fragment de code hostile, il faudrait construire du code autour afin que ce dernier n'apparaisse plus sous sa forme de départ i.e. il ne pourra pas être distingué dans la séquence résultante (code hostile $\cup$ séquence rajoutée). Le problème a été modélisé sous forme d'un CSP et une approche de construction de séquence de bytecode liant deux fragments de code a été présentée en détails. L'idée était de rajouter une ou plusieurs instructions bytecode avant le code hostile tout en respectant un ensemble défini de contraintes afin que le code résultant soit syntaxiquement et sémantiquement correct.

Cependant, comme déjà souligné auparavant (chapitre positionnement) le rajout d'une séquence de code (une ou plusieurs instructions bytecode) avant le code hostile a des effets sur le code original : celà provoque un décalage dans la séquence d'instructions interprétées, que nous avons appelé « désynchronisation du code ». La nouvelle séquence peut rester décalée jusqu'à ce que la séquence originale soit éventuellement retrouvée. En effet, en fonction du code à cacher et de la séquence rajoutée l'interprétation du code résultant va changer. Rappelons que le processeur commence par décoder l'octet représentant l'opcode d'une instruction afin de déterminer le nombre nécessaire d'octets à considérer comme opérandes et passer à l'instruction suivante. Donc, tout changement dans l'octet traité comme opcode provoque très probablement un changement dans l'interprétation de la séquence de code restante.

Ce principe de désynchronisation est illustré à travers l'exemple présenté ci-dessous (figure 5.1). Etant donné un code initial composé de deux instructions bytecode, `sinc` et `ret`, nous cherchons à le désynchroniser en rajoutant l'instruction `getfield_a`. Ainsi, le calcul du décalage de la suite des instructions initiales passe par les étapes suivantes :

- *Etape1* : l'instruction `getfield_ a`, qui a besoin d'un opérande, prend l'octet `0x59` correspondant à l'opcode de l'instruction initiale `sinc` comme étant son opérande.
- *Etape 2* : l'octet suivant de valeur `0x03` est considéré comme étant l'instruction suivante, `aconst_0` qui a zéro opérande.
- *Etape 3* : l'octet suivant de valeur `0x10` devient à son tour une instruction `bspush` qui a besoin d'un opérande. Ce dernier correspond à l'opcode de l'instruction initiale `ret` (de valeur `0x72`).
- *Etape 4* : le dernier octet de la séquence initiale, de valeur `0x3B`, est considéré comme étant une instruction `pop` qui n'a pas besoin d'opérande.

A la fin du calcul du décalage nous obtenons une nouvelle séquence de code, différente de l'initiale, regroupant quatre nouvelles instructions complètes. Ainsi, nous pouvons dire que nous avons appliqué une désynchronisation du code initial.

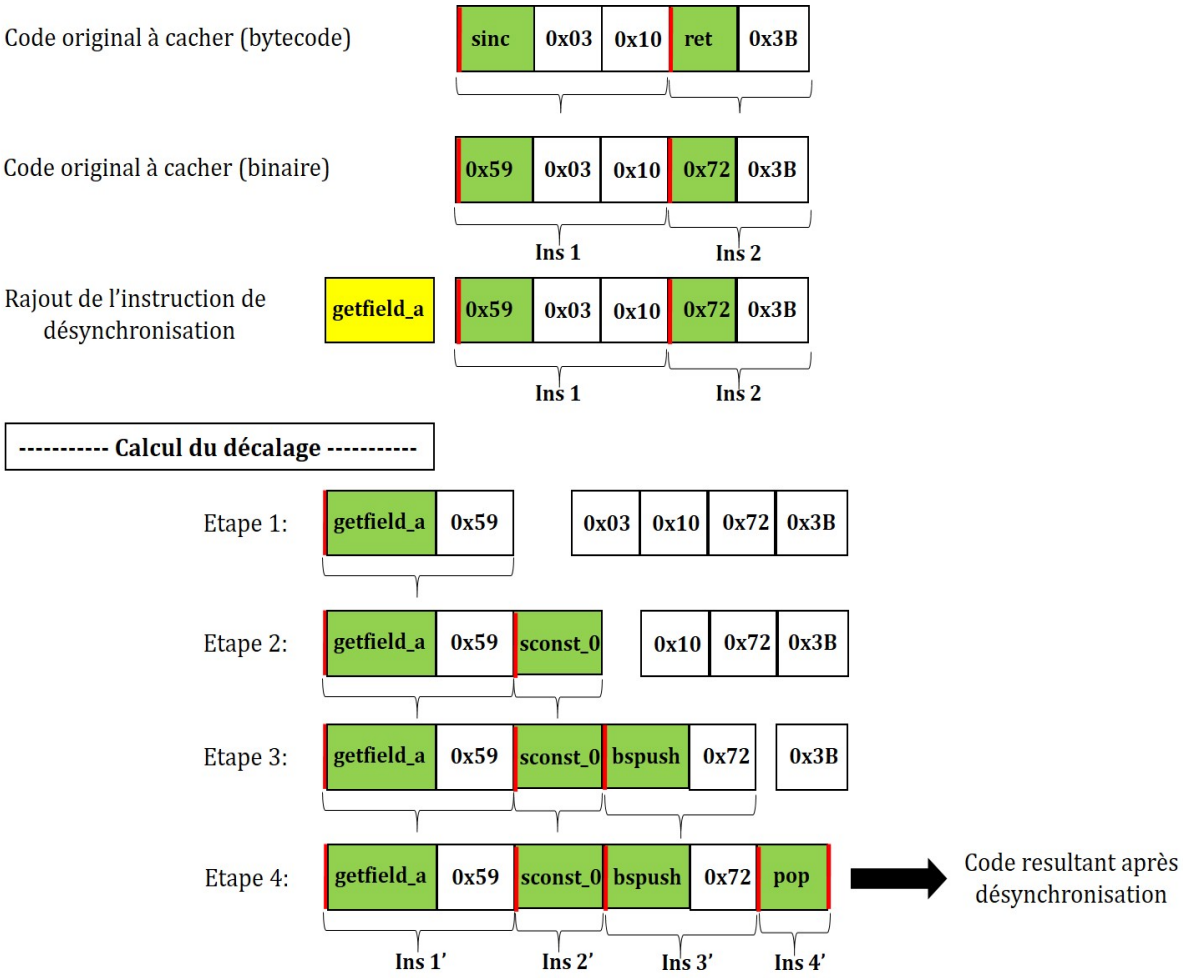

Fig 5.1 – Exemple de désynchronisation de code

5.2 Le modèle de désynchronisation

Rappelons que notre principal objectif est de dissimuler un code hostile afin qu'il puisse passer inaperçu. Ceci à travers le mécanisme de désynchronisation de code que nous avons défini. Afin de mieux cerner la problématique et ainsi apporter une solution adéquate, nous nous sommes posés un certain nombre de questions, auxquelles il faudrait répondre :

1. **Combien d'instructions devons-nous rajouter ?** une ou plusieurs instructions.
2. **Où devons-nous désynchroniser le code ?** juste avant le début du code hostile ou bien un peu plus en arrière (i.e. le code le précédant).
3. **Quel type d'instructions devons-nous rajouter ?** sans opérandes, avec 1 à 4 opérandes ou bien avec un nombre variable d'opérandes.
4. **Quelles parties de l'instruction devons-nous rajouter ?** l'instruction complète (opcode + tous ses opérandes), l'instruction partielle (opcode + quelques opérandes mais pas tous) ou bien juste l'opcode de l'instruction.

La réponse à chacune de ces questions va permettre de fixer un paramètre nécessaire pour l'application de la désynchronisation sur un code donné. Donc, au total nous aurons quatre paramètres à déterminer et qui constituent ce que nous appelons « le modèle de désynchronisation ». Le but de ce modèle est de permettre la caractérisation des effets de la dissimulation sur le code original (au niveau

bytecode) en essayant de faire apparaître tous les cas particuliers qui peuvent en découler.

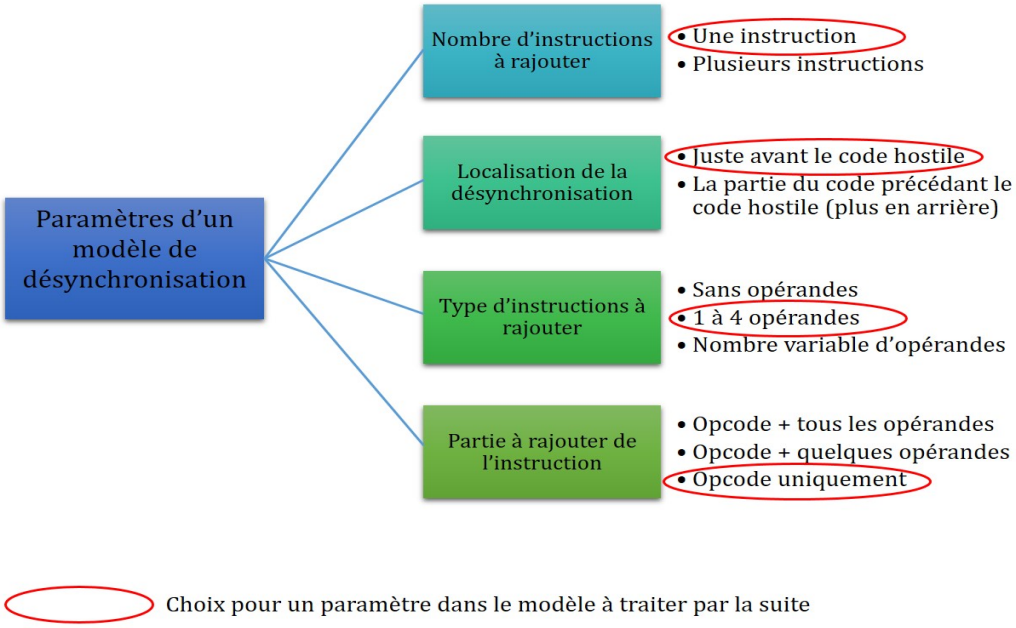

Fig 5.2 – Les paramètres d'un modèle de désynchronisation

Ainsi, pour cacher un code donné en appliquant la désynchronisation, il faudrait d'abord choisir un modèle de cette dernière. Ceci en fixant les paramètres relatifs aux questions précédemment posées (figure 5.2). Donc, au final plusieurs combinaisons sont possibles, ce qui implique aussi plusieurs modèles possibles.
Dans le cadre de cette thèse, nous allons nous intéresser à un modèle de désynchronisation spécifique en présentant une étude détaillée du mécanisme (i.e. le cas général ainsi que tous les cas particuliers qui en résultent). Toutefois, les autres modèles pourraient être traités en suivant la même démarche mais en tenant compte des spécificités de chacun d'entre eux.
Comme souligné dans la figure 5.2, le modèle sur lequel a porté notre étude est le suivant :

« Désynchroniser un code donné en rajoutant *l'opcode* d'*une seule* instruction dont le *nombre d'opérandes varie entre 1 et 4* et ceci *juste avant son début* ».

5.3 Preuve de concept : exemple de virus activable par attaque en faute

A travers l'exemple donné ci-dessous (listing 5.1), présenté dans [HL13], nous allons expliquer comment cacher un code malveillant dans un programme correct et inoffensif. L'objectif est de dissimuler l'appel d'une méthode qui obtient la valeur d'un conteneur de clé chiffrée (méthode *getKey()*).
En effet, le chargement du code donné dans le listing 5.1, tel qu'il est, sera refusé suite à une simple analyse du code. D'où la nécessité d'appliquer des transformations dessus afin qu'il apparaisse sous une autre forme inoffensive mais correcte. Une fois l'étape de vérification passée, le code original (appel à *getKey()* en clair) peut être retrouvé et exécuté moyennant une attaque par injection de faute.

```
public void process(APDU apdu) {
        /******************* Bloc B1 *************************/
         ... //local variables
        byte[ ] apduBuffer = apdu.getBuffer();//get the APDU Buffer
        if (selectingApplet()) {return;}
        byte readByte = (byte) apdu.setIncomingAndReceive();

        /******************* Bloc B2 *************************/
        DES_keys.getKey(apduBuffer, (short) 0);

        /******************* Bloc B3 *************************/
        apdu.setOutgoingAndSend((short) 0, DES_keys.getSize());
}
```

Listing 5.1 – Le code du virus

Ce code peut être divisé en trois blocs :

- Le bloc *B1* est un code correct qui doit être exécuté.
- Le bloc *B2* correspond au code qui doit être caché (le code hostile) et qui ne doit pouvoir être exécuté que suite à l'injection de la faute.
- Le bloc *B3* est un code correct qui doit être exécuté.

Le bytecode correspondant au code source précédent est donné dans le listing 5.2.

```
public void process(APDU apdu) {
/******************* Bloc B1 *************************/
        /*00bd*/ L0:  aload_1                 // apdu
        /*00be*/      invokevirtual   8       // getBuffer (APDU class)
        /*00c1*/      astore          4       //L4 = apduBuffer
        /*00c3*/      aload_0                 // this = Applet instance
        /*00c4*/      invokevirtual   9       // selectingApplet()
        /*00c7*/      ifeq            L1      // rel:+3 (@00ca)
        /*00c9*/      return
        /*00ca*/ L1:  aload_1                 // apdu
        /*00cb*/      invokevirtual   10
        /*00ce*/      s2b                     // readByte
        /*00cf*/      sstore          5       // L5 = readByte

/******************* Bloc B2 *************************/
        /*00d6*/      getfield_a_this 1       // DES_keys
        /*00d8*/      aload           4       // L4=>apdubuffer
        /*00da*/      sconst_0
        /*00db*/      invokeinterface   nargs : 3,index : 0 const: 3,
                              method: 4 //getkey
        /*00e0*/      pop                           // returned byte

/******************* Bloc B3 *************************/
        /*00e1*/      aload_1                 //L1 apdu
        /*00e2*/      sconst_0
        /*00e3*/      bspush 0x0F             // DES_keys size
        /*00e5*/      invokeinterface   nargs : 1,index : 0 const: 3,meth.: 1
```

```
        /*00ea*/      invokevirtual   11      // setOutgoingAndSend
        /*00ed*/      return
}
```

Listing 5.2 – Le code du virus au niveau bytecode

Dissimuler *B2*, et plus précisément l'instruction `invokeinterface`, consiste à insérer une instruction avant son début de manière à ce que les contraintes expliquées précédemment soient vérifiées (section 4.3.2). Mais avant de choisir une instruction à rajouter, les liens doivent être résolus statiquement. En effet, le processus final de l'édition des liens se fait à l'intérieur de la carte. Donc nous ne pourrions pas compter sur ce processus pour résoudre automatiquement les adresses. Par conséquent, nous nous sommes basé sur l'attaque présentée dans [HBL+12] qui permet de récupérer les informations de liaison. Ainsi, l'adresse de la méthode *getKey()* après édition des liens, pour la carte utilisée pour cette attaque, est `0x023C`. Donc, le bytecode correspondant à l'instruction `invokeinterface` devient comme indiqué dans le listing 5.3.

```
/*00db*/ invokeinterface 03 023c 04 // nargs : 3 , @023c , method : 4
/*00e0*/ pop                        // pop the return byte of the method
```

Listing 5.3 – Le code à cacher après résolution des liens

Parmi les instructions qui peuvent être sélectionnées pour dissimuler le code hostile, nous avons choisi dans cet exemple sur l'instruction `ifle` (son opcode est `0x65`). Elle utilise une valeur de type `short` et son opérande est un *offset* à l'instruction de branchement. Le fragment de code *B2* à charger dans la carte devient comme indiqué dans le listing 5.4. Ce dernier montre clairement le décalage provoqué par le changement des octets interprétés.
Considérons un modèle de faute « Erreur sur octet précis ». Après l'injection de la faute, la valeur de l'octet à l'offset `0x00db` devient `0x00`, ce qui correspond à une instruction `nop`. Ainsi le code *B2* original sera retrouvé et exécuté. Par conséquent, la clé secrète sera envoyée en clair au monde extérieur de la carte.

```
/*00db*/ [65]  ifle  @0x8D        // 0x8D corresponds to invokestatic
/*00dd*/ [03] sconst_0            // corresponds to the nargs
/*00de*/ [02] sconst m1               // corresponds to the address high
/*00df*/ [3c]  pop2                       // corresponds to the address low
/*00e0*/ [04] sconst_1            // corresponds to the method number
/*00e1*/ [3b] pop                         // resynchronized with the original code
```

Listing 5.4 – Le code hostile caché : résultat de la désynchronisation

Comme nous pouvons le voir dans cet exemple, cacher un code hostile consiste principalement à transformer un fragment du code en considérant qu'un opérande peut être exécuté comme une instruction et inversement. C'est ce que nous avons appelé *la désynchronisation de code*. Mais cette transformation doit prendre en compte plusieurs contraintes lors du choix des instructions à rajouter afin d'obtenir toujours un programme valide vis-à-vis de la spécification de la JCVM. De plus, plusieurs cas particuliers peuvent surgir suite à ce changement du code et nécessitent par conséquence des transformations supplémentaires.

5.4 Étude détaillée du mécanisme de désynchronisation

L'objectif de cette section est d'étudier tous les cas pouvant découler de l'application d'une désynchronisation de code et présenter les traitements nécessaires dans le but de retrouver, dans la mesure du possible, un code final correct après la transformation. Le modèle adopté est celui présenté dans la section 5.2 : ajouter l'opcode d'une seule instruction juste avant le début du code hostile.

5.4.1 Objectif : code correctement désynchronisé

D'abord, nous allons définir ce que nous avons qualifié de *code correctement désynchronisé*. Ce concept, qui reviendra dans tout le reste du chapitre, est illustré par la figure 5.3. Étant donné un code hostile à cacher figurant entre deux fragments de codes sains (c'est un cas général). Ce code hostile dispose de deux états mémoires : un état mémoire initial ($E_{initial}$) qui résulte de l'exécution du code sain le précédant et un état mémoire final (E_{final}) résultant de son exécution complète. L'ajout d'une instruction de désynchronisation [1] suivant le modèle choisi dégagera deux nouveaux états mémoire :

— E_1 : c'est l'état mémoire avant l'exécution de l'instruction de désynchronisation. Il est calculé à partir de l'état mémoire initial ($E_{initial}$) et la pré-condition de cette dernière.

— E_2 : c'est l'état mémoire résultant à la fin du décalage.

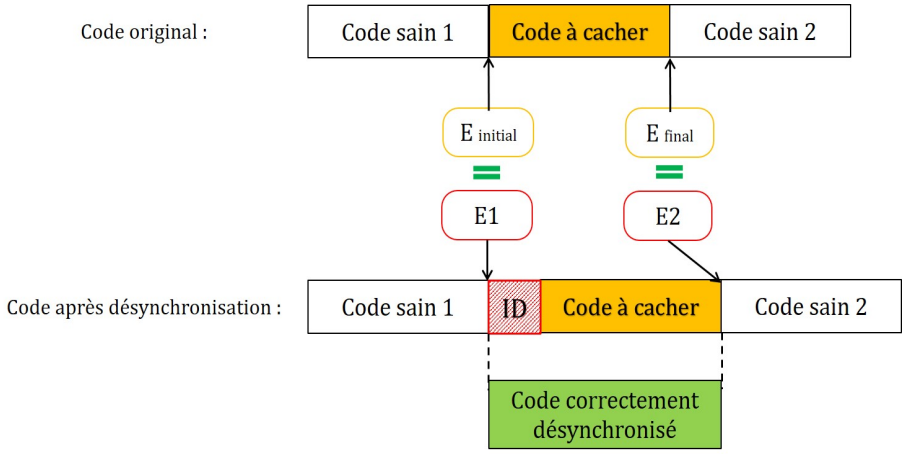

Fig 5.3 – Concept du code correctement désynchronisé

Un code correctement désynchronisé est un code où les états E_1 et E_2 sont égaux respectivement aux états originaux $E_{initial}$ et E_{final}. Autrement dit, la séquence désynchronisée pourrait être intégrée dans le code original sans poser des problèmes d'incompatibilité d'états mémoire au cours de l'exécution du code résultant. Notons qu'à ce stade nous faisons abstraction à l'étape de l'injection de faute qui va suivre cette étape de construction de code désynchronisés. Elle doit être traitée par la suite comme moyen de vérification des solutions obtenues à la fin de la construction.

5.4.2 Construction des codes correctement désynchronisés

Afin de traiter le problème de la désynchronisation de la façon la plus complète possible, nous avons suivi une démarche itérative (figure 5.4) qui regroupe principalement les étapes suivantes :

1. Commencer par traiter le problème de base : calculer tous les décalages possibles d'un code donné.

1. Dans la suite du document, nous utiliserons la notation *ID* pour désigner l'instruction de désynchronisation.

2. Mettre en place une implémentation fonctionnelle de ce noyau pour récupérer les premiers résultats expérimentaux.
3. Analyser les résultats obtenus pour tirer des conclusions sur les problèmes rencontrés lors de la construction des codes correctement désynchronisés et comment y répondre dans les itérations suivantes.
4. Le processus est répété jusqu'à ce qu'il n'y ait plus de problème qui puisse être traité.

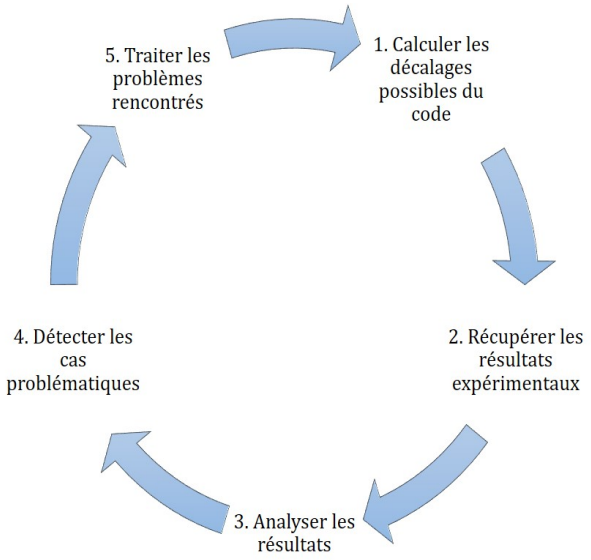

Fig 5.4 – Démarche suivie pour la construction des codes correctement désynchronisés

L'algorithme présenté dans cette section (algorithme 3) résume notre approche de base pour la construction de codes désynchronisés et qui sera amélioré dans les sections suivantes.

Partant d'un code hostile à cacher (i.e. à désynchroniser), les deux états mémoires lui correspondant $E_{initial}$ et E_{final} ainsi que le jeu d'instructions bytecode. Nous cherchons à obtenir toutes les combinaisons possibles d'une désynchronisation correcte de ce code (i.e. conformes à la définition d'une solution correcte donnée dans la section 5.4.1).
Les instructions de décalage sont organisées en quatre catégories en fonction de leur nombre d'opérandes : un, deux, trois ou quatre opérandes. Le calcul du décalage est fait de la même façon pour les quatre catégories. Selon le modèle de désynchronisation considéré, une instruction ID [2] est rajoutée au début du code hostile. L'état mémoire la précédant (E_1) est calculé pour commencer l'interprétation de la nouvelle séquence de code (ID+code hostile) s'il est égal à $E_{initial}$. Le cas échéant, cette séquence est parcourue en interprétant successivement les octets rencontrés (instructions ou opérandes) tout en se référant au jeu d'instructions donné en entrée. Un état mémoire est calculé pour chaque instruction. De plus, les contraintes définies dans la section 4.3.2 doivent être respectées à chaque étape de l'interprétation. A la fin de la séquence, l'état mémoire E_2 est récupéré pour procéder à sa comparaison à l'état mémoire E_{final}. S'il y a égalité, la séquence est considérée comme étant une solution correcte (i.e.un code correctement désynchronisé) et sauvegardée. Sinon, la séquence est sauvegardée comme étant un code problématique à traiter ultérieurement. Le processus est répété pour toutes les instructions des quatre catégories. A la fin deux ensembles de codes sont restitués :

2. D'après notre modèle de désynchronisation, nous sous entendons par l'instruction ID son opcode uniquement. Ceci est valable pour tout le reste du chapitre

Algorithme 3 : Construction de codes désynchronisés

Entrées :
- Un code hostile à désynchroniser
- Un ensemble d'instructions bytecode
- Un état mémoire initial ($E_{initial}$)
- Un état mémoire final (E_{final})

Sorties :
- Un ensemble de codes correctement désynchronisés
- Un ensemble de codes problématiques

```
1  Début
2      Pour i = 1 à 4 faire
3          Pour chaque instruction ID ∈ {catégorie_i des instructions bytecode} faire
4              - Rajouter l'instruction ID au début du code hostile ;
5              - Calculer l'état mémoire avant l'instruction ID (E_1) ;
6              si E_1 = E_initial alors
7                  - Interpréter le nouveau code (ID + codeHostile) en calculant le décalage (un
                     état mémoire par instruction interprétée) ;
8                  - Récupérer l'état final après le décalage (E_2);
9                  - Analyser la solution obtenue i.e. comparer (E_2, E_final) ;
10                 si Solution correcte alors
11                     - Mémoriser la solution trouvée ;
12                 sinon
13                     - Mémoriser le code problématique ;
14                 fin
15             sinon
16                 - Mémoriser le code problématique ;
17             fin
18         finprch
19     finpr
20     - Etudier les codes problématiques et détecter les catégories des problèmes possibles de la
         désynchronisation ;
21 Fin
```

- *Ensemble 1* : Toutes les solutions correctes trouvées .
- *Ensemble 2* : Tous les codes problématiques dans le but de les étudier pour dégager les classes possibles des problèmes liés à la désynchronisation de code. Ceci afin d'y remédier en trouvant les traitements adéquats pour augmenter le nombre de combinaisons possibles.

5.4.3 Étude des cas problématiques

Suite à la récupération des résultats expérimentaux fournis par le noyau implémentant l'algorithme 3, nous avons procédé à l'analyse des solutions classées comme problématiques. Nous avons pu dégagé deux grandes catégories des problèmes pouvant surgir lors de la construction d'un code correctement désynchronisé :

- Problèmes liés à la structure du code.
- Problèmes liés aux états mémoire.

5.4.3.1 Cas 1 : Problèmes liés à la structure du code

Ce sont des problèmes rencontrés lors de l'interprétation de la nouvelle séquence (instruction *ID*+ code hostile). Ils sont liés à la composition de cette séquence. Deux problèmes sont distingués dans cette catégorie :

— **Cas 1.1. Rencontrer un opcode non valide :** L'opcode d'une instruction bytecode est un code hexadécimal qui prend une valeur allant de `0x00` à `0xB8` (section 4.3.1). Lors de l'interprétation de la séquence de code, la valeur d'un octet considéré comme étant l'opcode d'une nouvelle instruction peut être une valeur qui n'appartient pas à l'intervalle défini. Ainsi, il est considéré comme étant un opcode non valide et l'interprétation du reste de la séquence ne pourrait pas être poursuivie.
La figure 5.5 montre un exemple qui illustre ce type de problème. L'instruction *ID* a besoin d'un opérande qui correspond à `ins1`. Donc, son premier opérande `op1` devient une instruction (opcode valide) qui prend par exemple à son tour deux opérandes qui sont `op2` et `ins2`. Par la suite, l'opérande `op1` (celui de `ins2`) est considéré comme une nouvelle instruction. Mais comme la valeur de cet octet ne correspond pas à un opcode valide, l'interprétation de la suite de la séquence est arrêtée.

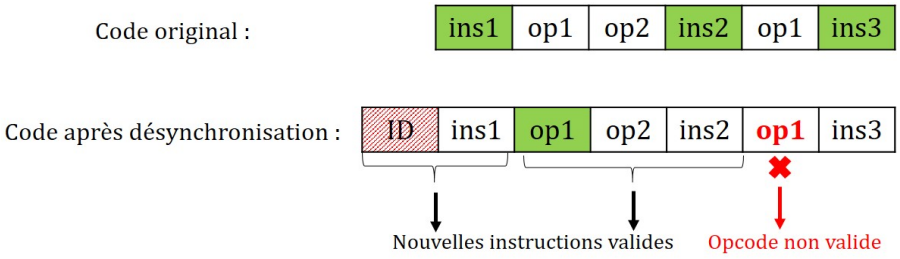

Fig 5.5 – Rencontrer un opcode non valide (Cas 1.1)

— **Cas 1.2. Manque d'opérandes à la fin de la séquence :** C'est un problème rencontré à la fin du calcul du décalage. La dernière instruction de la séquence ne trouve pas tous les opérandes qui lui sont nécessaires dans la séquence interprétée. Dans l'exemple de la figure 5.6, la dernière instruction `op1` (opérande de `ins2` précédemment) à besoin de deux opérandes qui ne sont pas disponibles dans la séquence traitée.

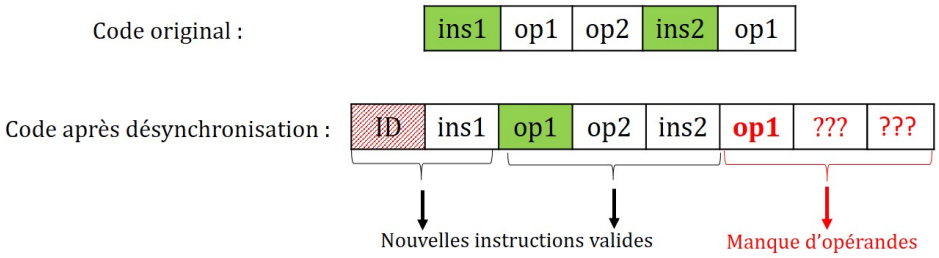

Fig 5.6 – Manque d'opérandes à la fin de la séquence (Cas 1.2)

5.4.3.2 Cas 2 : Problèmes liés aux états mémoire

Il s'agit des problèmes liés à la compatibilité d'un état mémoire (pile des opérandes et liste des variables locales) courant avec l'état mémoire relatif à l'instruction qui va être exécutée. La compatibilité concerne le type des éléments manipulés et la taille de la pile. Un problème de cette catégorie peut surgir à trois niveau différents de la séquence interprétée (figure 5.7) :

— **Cas 2.1. Au début :** La pré-condition de l'instruction *ID* est incompatible avec l'état mémoire initial ($E_1 \neq E_{initial}$).

— **Cas 2.2. Au milieu :** La pré-condition de l'instruction courante est incompatible avec l'état mémoire précédent.

— **Cas 2.3. A la fin :** L'état mémoire à la fin du décalage est incompatible avec l'état mémoire final désiré ($E_2 \neq E_{final}$).

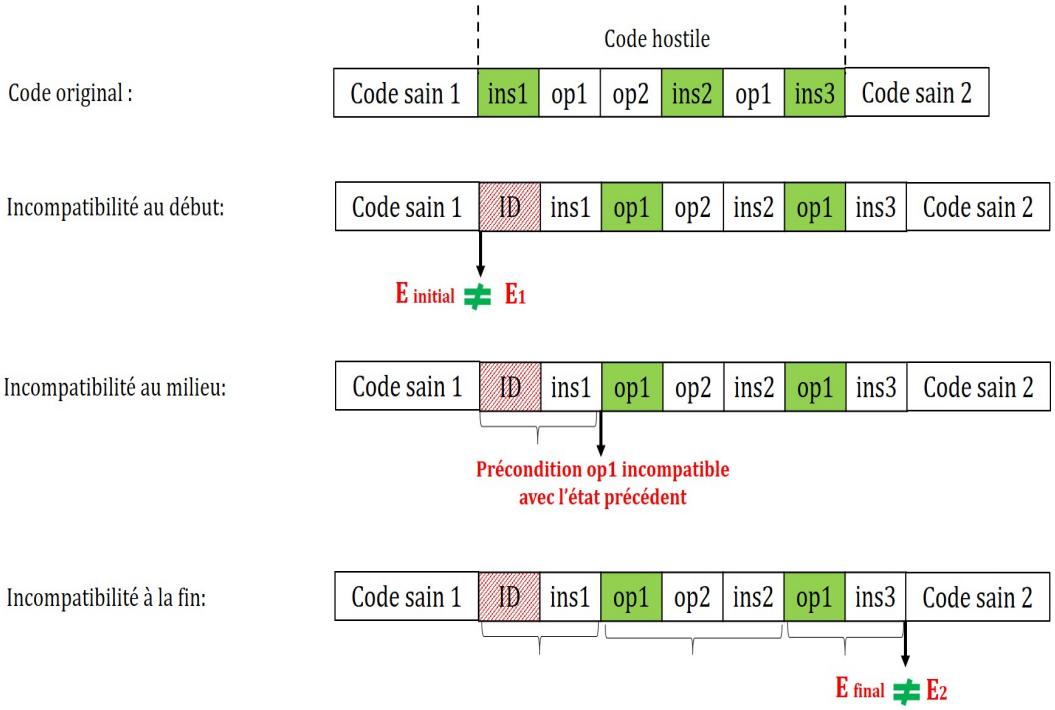

Fig 5.7 – Cas 2. Incompatibilité des états mémoire

5.5 Traitement des cas problématiques

5.5.1 Rajout de préambule

Dans cette section, nous allons traiter le problème d'incompatibilité de l'état mémoire au début de la séquence (cas 2.1). Le problème qui se pose est que l'état mémoire avant l'instruction *ID* (E_1) est différent de l'état initial ($E_{initial}$) considéré au début du code hostile. L'idée pour résoudre ce problème est de rajouter une suite d'instructions bytecode, que nous appelons « *préambule* », avant l'instruction *ID* de telle façon à converger vers l'état mémoire désiré i.e. $E_{initial}$. Autrement dit, nous cherchons à lier les deux états mémoire E_1 et $E_{initial}$ en trouvant des chemins possibles tout en raisonnant dans le sens inverse que celui de l'exécution (figure 5.8). Pour la construction du *préambule* nous allons appliquer notre approche de construction de séquences de code, présentée dans le chapitre 4, dans le but d'avoir toutes les solutions possibles i.e. les séquences qui peuvent être rajoutées comme préambule.

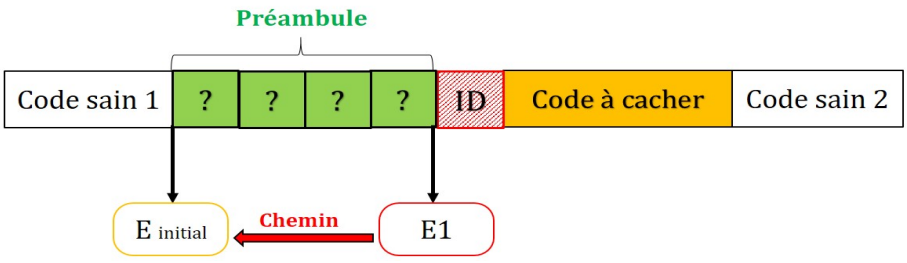

Fig 5.8 – Rajout de préambule

L'idée est de faire la génération/exploration d'un arbre de recherche où :
— La racine de l'arbre correspond à l'instruction *ID* à laquelle est associée à un état mémoire de départ E_1.
— Les nœuds intermédiaires de chaque niveau de l'arbre représentent les instructions précédentes possibles.
— Les feuilles de l'arbre correspondent à l'état mémoire final à atteindre $E_{initial}$.
— La profondeur maximale dans l'arbre correspond à la taille maximale du préambule qui pourrait être rajouté avant l'instruction ID.

Tous les chemins de la racine vers les feuilles représentent des séquences d'instructions qui peuvent être rajoutées avant l'instruction *ID* i.e. un préambule possible.

5.5.2 Rajout de postambule

Dans cette section, nous allons présenter une solution similaire à la précédente afin de traiter deux autres cas problématiques. Il s'agit du rajout d'un fragment de code à la fin du code désynchronisé (après calcul du décalage), c'est ce que nous appelons « *postambule* ». Nous distinguons deux types de postambules, chacun résout un cas précis.

5.5.2.1 Postambule-type-1

Son objectif est de résoudre le problème du manque d'opérandes à la fin de la séquence désynchronisée (cas 1.2 - section 5.4.3.1). Il s'agit de compléter la dernière instruction de la séquence par les opérandes qui lui sont nécessaires. Ces opérandes rajoutées sont appelées « *Postambule-type-1* ».

5.5.2.2 Postambule-type-2

Son objectif est de résoudre le problème d'incompatibilité de l'état mémoire à la fin de la séquence désynchronisée (cas 2.1 - section 5.4.3.2). Le problème qui se pose est que l'état mémoire à la fin du décalage (E_2) est différent de l'état final (E_{final}) déterminé à la fin du code hostile. L'idée pour résoudre ce problème est de rajouter une suite d'instructions bytecode, que nous appelons « *postambule-type-2* », après la séquence désynchronisée de telle façon à converger vers l'état mémoire désiré i.e. E_{final}. Autrement dit, nous cherchons à lier les deux états mémoire E_2 et E_{final} en trouvant des chemins possibles tout en raisonnant cette fois-ci dans le sens de l'exécution (figure 5.9). Pour la construction du *postambule* nous allons adapter notre approche de construction de séquences de code dans le but d'avoir toutes les solutions possibles i.e. les séquences qui peuvent être rajoutées comme postmbule.

L'idée est de faire la génération/exploration d'un arbre de recherche où :
— La racine de l'arbre correspond à la dernière instruction du décalage et à laquelle est associé un état mémoire de départ E_2.
— Les nœuds intermédiaires de chaque niveau de l'arbre représentent les instructions suivantes possibles.
— Les feuilles de l'arbre correspondent à l'état mémoire final à atteindre E_{final}.
— La profondeur maximale dans l'arbre correspond à la taille maximale du postambule qui pourrait être rajouté à la fin de la séquence désynchronisée.

Tous les chemins de la racine vers les feuilles représentent des séquences d'instructions qui peuvent être rajoutées après la séquence désynchronisée i.e. un postambule possible.

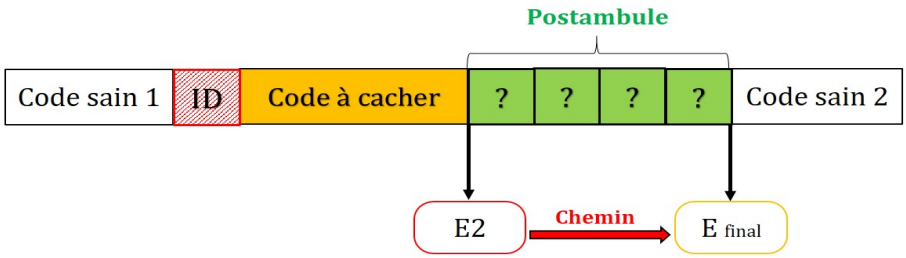

Fig 5.9 – Rajout de postambule-type-2

5.5.3 Cas abandonnés

Dans les sections précédentes, nous allons présenté les traitements qui peuvent être adoptés pour résoudre trois cas problématiques parmi ceux listés dans la section 5.4.3. Cependant, il reste les deux cas suivants pour lesquels aucune solution n'a pu être trouvée (voir même elle est inexistante) :

- *Rencontrer un opcode non valide (Cas 1.1).* Comme le jeu d'instructions bytecode est un ensemble fini et les valeurs des opcodes correspondant à des instructions valides sont connues, toute valeur n'appartenant pas à cet ensemble ne peut être considérée comme étant un opcode correspondant à une instruction valide. Donc un tel cas est abandonné sans traitement.
- *Incompatibilité d'état mémoire au milieu de la séquence désynchronisée (cas 2.2).* Nous avons montré que l'incompatibilité des états mémoire au début ou à la fin de la séquence à désynchroniser pourrait être traitée en rajoutant une séquence de code au début ou bien à la fin de cette dernière. Cependant, une incompatibilité au milieu de la séquence à désynchroniser (le code hostile) ne peut pas être résolue du moment que le contenu de cette dernière est fixe (le code hostile est une suite d'octets spécifiques) et donc ne peut subir aucun changement mais juste le décalage de son interprétation. Ceci dit, un tel cas est aussi un cas abandonné sans traitement.

5.6 Récapitulatif de l'étude

A travers l'étude présentée dans ce chapitre, le problème de désynchronisation a été traité dans sa totalité pour un modèle bien défini. Dans le cas général (section 5.4.1), un code correctement désynchronisé a été défini comme étant un code qui préserve les états mémoires avant son début et à sa fin après le rajout de l'instruction ID juste avant son début. Autrement dit, cette dernière peut être rajoutée sans affecter l'exécution normale du code (aboutit au même résultat). Cinq cas problématiques ont été détectés (section 5.4.3), dont trois ont pu être résolus tandis que les deux autres sont abandonnés. L'ensemble de ces problèmes est résumé dans le tableau 5.1.

5.7 Conclusion

Arrivés à la fin de chapitre, nous avons pu répondre à la problématique principale soulevée au début de ce travail : « *Comment construire un code malveillant activable par attaque en faute ?* ». Nous avons traité un modèle précis de désynchronisation en présentant tous les cas de figure qui peuvent survenir. La solution proposée se base en partie sur l'approche de construction de séquence de code traitée dans le chapitre 4 pour résoudre deux cas problématiques détectés dans notre étude : l'ajout du préambule et l'ajout du postambule.
Dans le chapitre suivant, nous allons présenter la mise en pratique de nos deux contributions à travers

Catégorie du problème	Problème	Traitement du problème
Problèmes liés à la structure du code	Rencontrer un opcode non valide au cours du calcul du décalage	Cas abandonné
	Manque d'opérandes à la fin de la séquence	Rajouter un postambule-type-1 (section 5.5.2.1)
Problèmes liés aux états mémoire	Incompatibilité de l'état mémoire au début de la séquence	Rajouter un préambule en se basant sur l'approche de construction de séquence de code par parcours d'arbre (section 5.5.1)
	Incompatibilité de l'état mémoire au milieu de la séquence	Cas abandonné
	Incompatibilité de l'état mémoire à la fin de la séquence	Rajouter un postambule-type-2 en se basant sur une adaptation de l'approche de construction de séquence de code par parcours d'arbre (section 5.5.2.2)

Tab 5.1 – Récapitulatif des cas particuliers rencontrés lors de la construction des codes désynchronisés

les deux outils développés ainsi que les cas d'étude illustrant les résultats expérimentaux qui ont été obtenus.

CHAPITRE 6

Implémentation, exploitations et détection

Sommaire

L'objectif de ce chapitre est de présenter la mise en œuvre des deux approches proposées dans le chapitre 4 (construction de séquence de code par parcours d'arbre) et le chapitre 5 (la désynchronisation de code). Pour chacune, nous présenterons l'architecture et le principe de fonctionnement de l'outil qui a permis son automatisation. En outre, des exemples illustrant les résultats générés sont donnés pour avoir une idée sur les exploitations possibles de nos approches. Par la suite, une étude de cas détaillée montre comment ces deux dernières peuvent être utilisées afin de rejouer une attaque existante. Nous soulignons le fait que les résultats restitués par les deux outils doivent subir une phase de manipulation/vérification du fichier CAP, tel que expliqué dans la section 4.5.4, avant qu'il ne soit possible de les charger dans une carte à puce réelle. A la fin, nous présenterons quelques outils existants pour la détection des vulnérabilités des injections de fautes, leurs limites ainsi que des pistes de solutions possibles.

6.1 Implémentation 1 : l'outil « Trace Generator »

L'approche de construction de séquence de code présentée dans le chapitre 4 est mise en œuvre à travers notre outil nommé *Trace Generator* (développé en Java). Il prend plusieurs entrées, effectue la génération/exploration de l'arbre de recherche (la version optimisée de l'approche, voir section 4.6.3) et restitue un fichier texte contenant toutes les solutions trouvées i.e. les chemins dans l'arbre correspondant à des séquences de code possibles. Dans ce qui suit, nous allons présenter quelques détails relatifs à cet outil.

6.1.1 Les modes de génération des solutions

Trace Generator offre deux modes de génération possibles : classique ou aléatoire.

— *Mode classique* : La génération de toutes les solutions possibles se fait à travers un parcours de l'arbre en profondeur (avec définition d'une profondeur maximale comme critère d'arrêt). Pour un nœud donné, la sélection des nœuds fils à explorer se base sur les heuristiques présentées dans la section 4.6.2 (bi-grams et tri-grams)). Dans ce mode, tous les nœuds fils du nœud courant doivent être visités avant d'explorer un autre nœud au même niveau dans l'arbre.

— *Mode aléatoire* : Ce mode est aussi basé sur les heuristiques pour trouver les instructions candidates (nœud fils). Cependant, l'ordre de sélection des fils à explorer est aléatoire. De plus, il faudrait à la racine de l'arbre après avoir trouvé un certain nombre de solutions (la valeur du nombre maximal de solutions avant le retour à la racine est fixée à priori). Il s'agit d'un redémarrage de la génération de l'arbre à partir de sa racine. Cela permet d'augmenter la diversité des solutions générées, i.e. que les solutions produites successivement sont très variées les unes des autres. Cette caractéristique est utile s'il y a besoin de générer un nombre réduit de solutions avec un haut niveau de diversité, même avec une grande profondeur dans l'arbre.

En résumé, en fonction du besoin, le mode classique peut être utilisé pour générer toutes les solutions possibles en sélectionnant les meilleurs fils d'abord. Tandis que le mode aléatoire est adopté pour générer un ensemble partiel de solutions, mais avec une plus grande diversité entre deux solutions successives.

6.1.2 Architecture de l'outil

L'outil *Trace Generator* nécessite trois entrées principales :

1. La liste des instructions bytecode représentée dans un format spécial (figure 6.2) pour faciliter l'extraction des contraintes spécifiques (section 4.3.2) pendant la génération/exploration de l'arborescence.
2. Les fichiers de statistiques bigrams et trigrams obtenus.
3. Les données de configuration, regroupant :
   — Le mode de génération (classique ou aléatoire).
   — L'instruction de départ (correspondant au nœud racine).
   — La profondeur maximale, i.e. le nombre de niveaux dans l'arbre (correspondant à la longueur maximale de la solution).
   — Le nombre maximal de solutions.
   — L'état mémoire de départ (resp. l'état mémoire d'arrivée). représenté par l'état de la pile d'opérandes et la liste des variables locales.
   — Le nombre de solutions avant le retour à la racine si le mode choisi est le mode aléatoire.

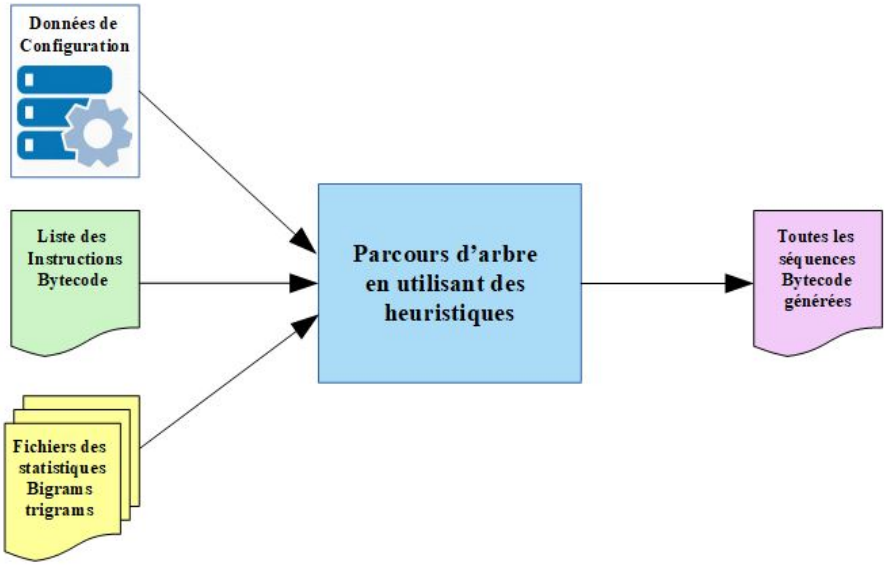

Fig 6.1 – Architecture générale de l'outil "Trace Generator"

La sortie de l'outil est un fichier texte contenant toutes les solutions trouvées i.e. tous les chemins de la racine aux feuilles de l'arbre. Ces solutions font l'objet d'une phase de vérification, tel que expliqué dans la section 4.5.4, afin de décider si elles sont acceptées ou non.
Notons que la version actuelle de l'outil gère un sous-ensemble significatif des instructions bytecode Java Card qui regroupe :

- Les instructions de manipulation de la pile d'opérandes et la liste des variables locales,
- Opérations arithmétiques,
- Opérations logiques,
- Opérations de manipulation des objets et tableaux,
- Instructions de branchement,
- Opérations de retour des méthodes.

Cependant, les instructions ci-dessous ne sont pas actuellement intégrées à l'outil (le jeu d'instructions est à compléter) :

- Instructions d'invocation de méthodes (`invokespecial`, `invokestatic`, `invokevirtual` et `invokeinterface`),
- Instructions à nombre variable d'opérandes (`stableswitch`, `slookupswitch`, `itableswitch`, `ilookupswitch`),
- Opérations d'exception (`athrow`).

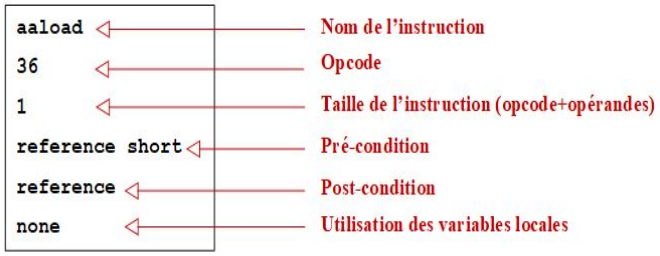

Fig 6.2 – Représentation d'une instruction bytecode dans le fichier d'entrée

6.1.3 Exemple d'application : une autre variante pour cacher getkey()

Nous allons reprendre l'exemple présenté dans la section 5.3 afin d'exposer une autre variante possible pour cacher l'appel à la méthode *getkey()* (désynchronisation du code) et illustrer l'utilité de l'outil *Trace Generator*.
Pour cela, nous avons pensé à une instruction qui n'a pas un effet sur l'état de la mémoire. Le choix a porté sur l'instruction `goto_w` dont la spécification est donnée ci-dessous (figure 6.3).

goto_w
Branch always (wide index)

Format
goto_w branchbyte1 branchbyte2

Forms
goto_w = 168 (0xa8)

Stack
No change

Description
The unsigned bytes branchbyte1 and branchbyte2 are used to construct a signed 16 bit branchoffset, where branchoffset is $(branchbyte1 \ll 8)|branchbyte2$. Execution proceeds at that offset from the address of the opcode of this goto instruction. The target address must be that of an opcode of an instruction within the method that contains this goto instruction.

Fig 6.3 – Spécification de l'instruction «*goto_w*» [Ora15b]

C'est une instruction qui prend deux opérandes qui sont utilisés pour calculer un offset qui doit être l'adresse d'une autre instruction figurant dans la même méthode que l'instruction de saut. Ainsi, le saut provoqué par l'instruction `goto_w` doit être géré afin d'avoir un code correct vis-à-vis de la spécification Java Card (figure 6.4).

Comme présenté dans la figure 6.5, l'idée est la suivante :

— A l'adresse du saut, placer une autre instruction `goto_w` avec un offset qui assure un saut en arrière ciblant l'instruction juste après le premier `goto_w`.

— Connaissant les deux états mémoires EM1 et EM2, il faudrait compléter le fragment de code entre l'adresse du saut et la fin de la méthode (dans le sens inverse que celui de l'exécution). C'est à ce niveau qu'intervient notre outil de génération de séquences de code.

— Le code résultant peut être chargé dans une carte sans qu'il ne soit détecté comme un code hostile vu que l'appel à la méthode *getKey()* a été dissimulé. Ce dernier peut être retrouvé après une injection de faute ciblant l'octet correspondant à la première instruction `goto_w`. Ainsi, comme le montre la figure 6.6, le fragment de code rajouté précédemment devient un code mort qui ne sera jamais exécuté (il est situé après le `return` de la méthode). Il a servi uniquement à cacher le code hostile.

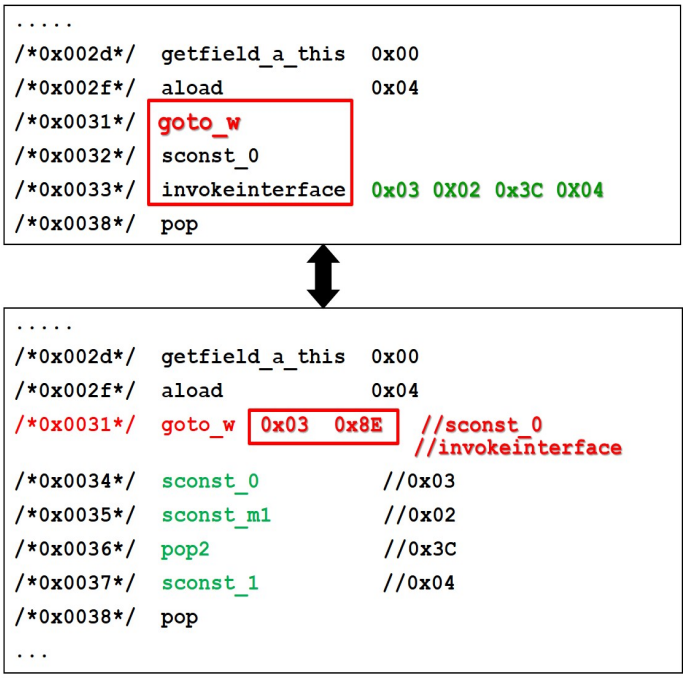

Fig 6.4 – Désynchronisation du code par l'ajout de l'instruction goto_w (1)

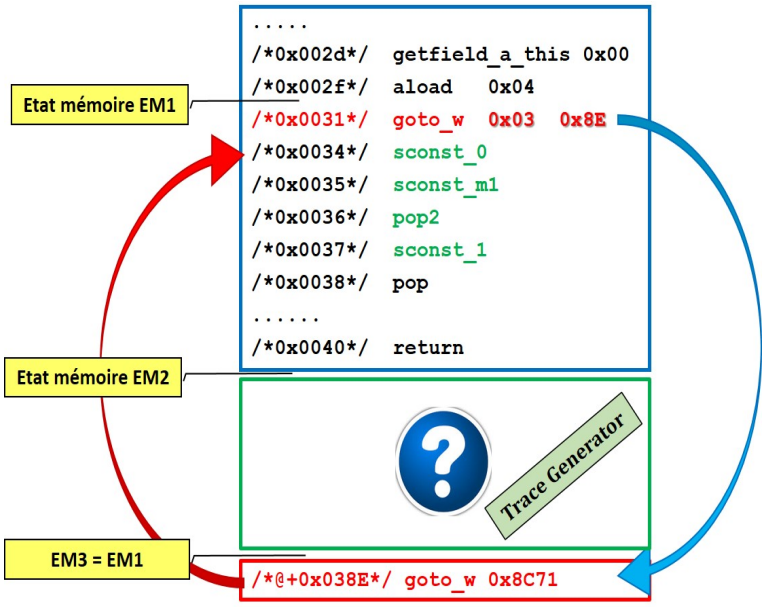

Fig 6.5 – Désynchronisation du code par l'ajout de l'instruction goto_w (2)

6.2 Implémentation 2 : l'outil de désynchronisation

L'outil présenté dans cette section est la mise en pratique de l'approche de désynchronisation de code présentée dans le chapitre 5. Il prend plusieurs entrées, calcul tous les cas possibles de désynchronisation d'une séquence de code à cacher (traite le cas général et les cas problématiques). Par la suite, il restitue tous les codes correctement désynchronisés possibles. Dans ce qui suit, nous allons présenter quelques détails relatifs à cet outil.

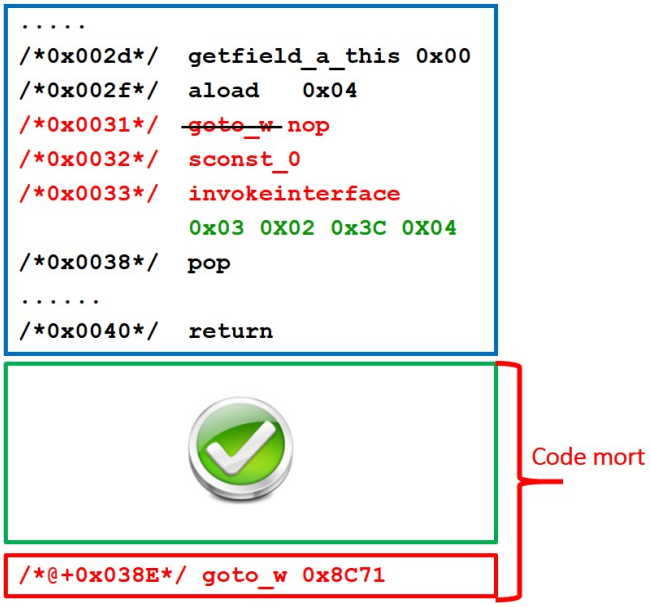

Fig 6.6 – Code hostile retrouvé après l'injection de la faute

6.2.1 Architecture de l'outil

L'outil de désynchronisation nécessite quatre entrées principales :

1. La liste des instructions bytecode représentée dans le même format que dans la figure 6.2.
2. La séquence de code à cacher (à saisir à travers l'interface ou charger directement à partir d'un fichier texte externe).
3. Les fichiers de statistiques bigrams et trigrams.
4. Les données de configuration, regroupant :
   - L'état mémoire de départ [1].
   - Les paramètres supplémentaires pour le calcul du préambule et postambule en cas de besoin :
     - La profondeur maximale, i.e. le nombre de niveaux dans l'arbre (correspondant à la longueur maximale de la solution).
     - Le nombre maximal de solutions.

L'architecture globale de l'outil est présentée dans la figure 6.7. Nous retrouvons quatre modules qui interagissent entre eux pour fournir des solutions. Le module *calcul du décalage* est la partie principale de l'outil, c'est la mise en œuvre du principe de la désynchronisation tel que défini dans le chapitre précédant (suivant le modèle choisi). Il fait appel aux deux autres modules (préambule et postambule) afin de traiter les cas problématiques. Ces deux modules sont inspirés de l'outil *Trace Générator* (mais fonctionnent dans deux sens inverses).

A la fin du processus de calcul du décalage un ensemble des codes correctement désynchronisés est fourni en sortie (voir section 6.2.2).

Le dernier module *injection de faute* est complémentaire aux précédents. Comme son nom l'indique, il simule l'injection d'une faute, selon le modèle de faute adopté (chapitre positionnement), sur les codes fournis en sortie par le module *calcul du décalage*. Et restitue à son tour la liste des codes désynchronisés qui ont résisté à l'injection de la faute i.e. les codes à cacher qui ont été transformés par la désynchronisation et retrouvés suite à une injection de faute.

1. L'état d'arrivée n'est pas fourni en entrée, vu que nous avons la séquence de code à désynchroniser, mais calculé lors du calcul du décalage et peut varier selon le cas traité

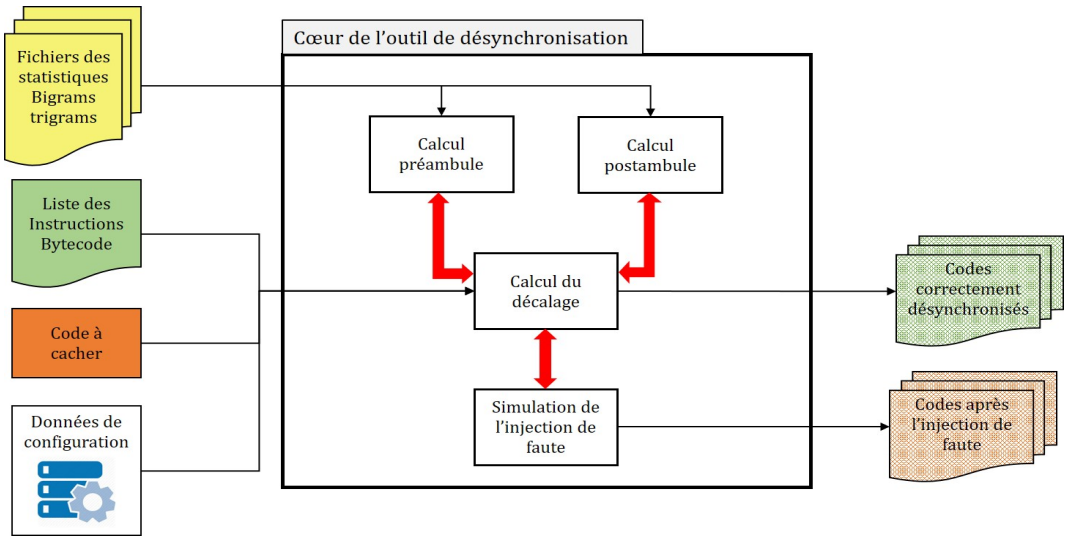

Fig 6.7 – Architecture générale de l'outil de désynchronisation

6.2.2 Exemples de sorties fournies par l'outil

Les résultats restitués par l'outil sont organisés en un ensemble de fichiers selon leur catégorie, plusieurs cas sont distingués (l'arborescence [2] présentée dans la figure 6.8).

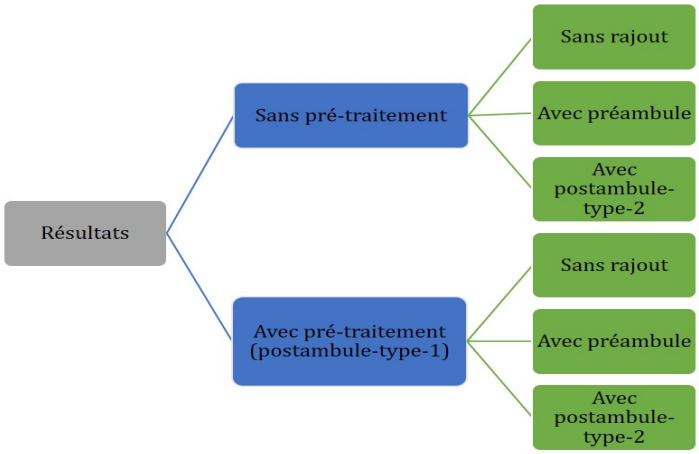

Fig 6.8 – Organisation des résultats de la désynchronisation

Rappelons que l'approche proposée pour la désynchronisation de code est applicable sur n'importe quel code indépendamment de sa nature (hostile ou non) i.e. il est traité comme étant une simple séquence de code à transformer. Ceci dit, l'exemple présenté ci-dessous est donné à titre illustratif uniquement (il n'a pas une signification particulière).

Entrées de l'outil :

— Code à désynchroniser : `|bspush 3| sinc 2 17| putstatic_b 4 163|`
— Etat de la pile initiale : `short, reference`
— Etat des variables locales : `reference, short, returnAddress`
— Pour les pré/postambules : Profondeur maximale=3, nombre de solutions max=100

2. pré-traitement : correspond au rajout du postambule-type-1 pour régler le problème du manque d'opérandes à la fin de la séquence

Ci-dessous, nous allons présenter quatre exemples des solutions générées pour l'exemple en entrée. Chaque figure annotée illustre une solution en sortie.

Exemple de solution "sans pré-traitement et sans rajout" :

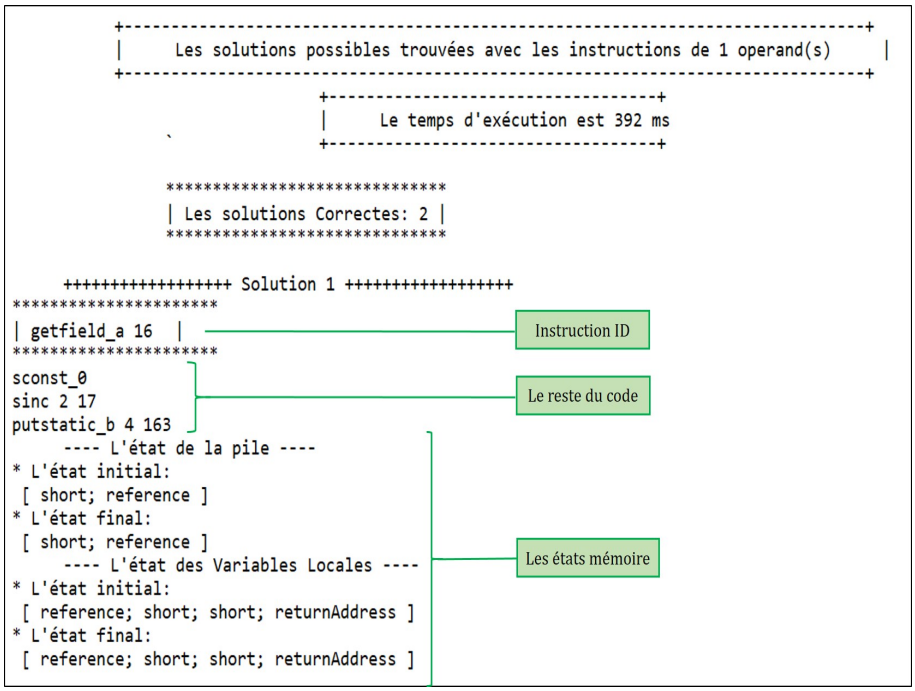

Fig 6.9 – Exemple de solution "sans pré-traitement et sans rajout"

Exemple de solution "avec pré-traitement et sans rajout" :

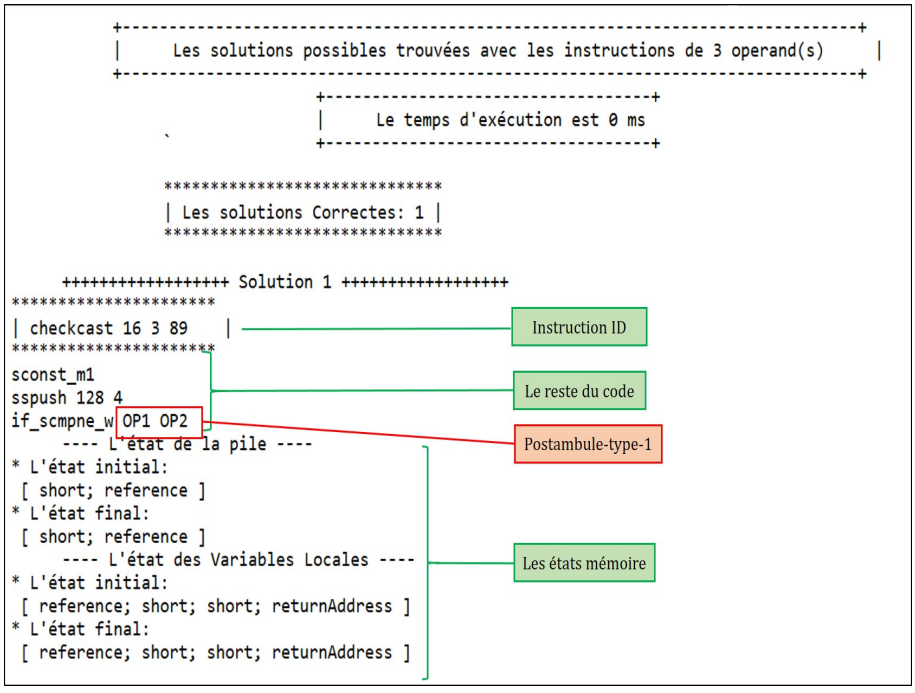

Fig 6.10 – Exemple de solution "avec pré-traitement et sans rajout"

Exemple de solution "avec préambule" :

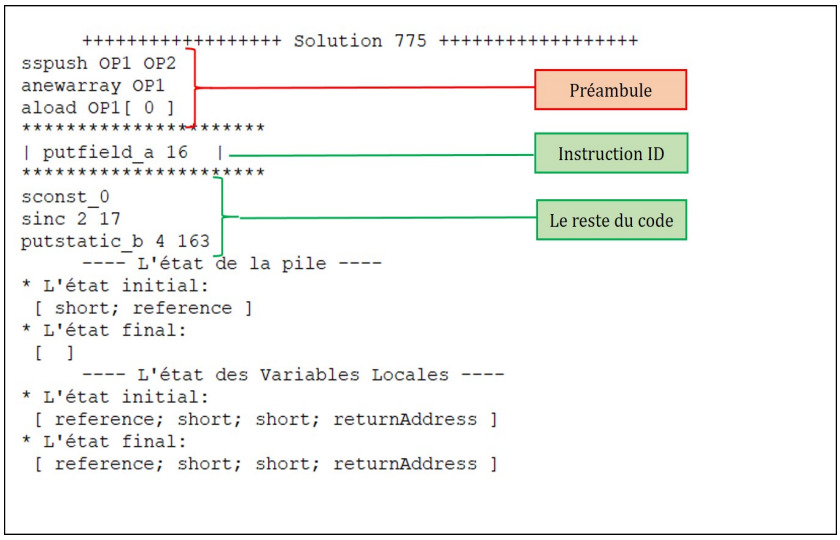

Fig 6.11 – Exemple de solution "avec préambule"

Exemple de solution "avec postambule" :

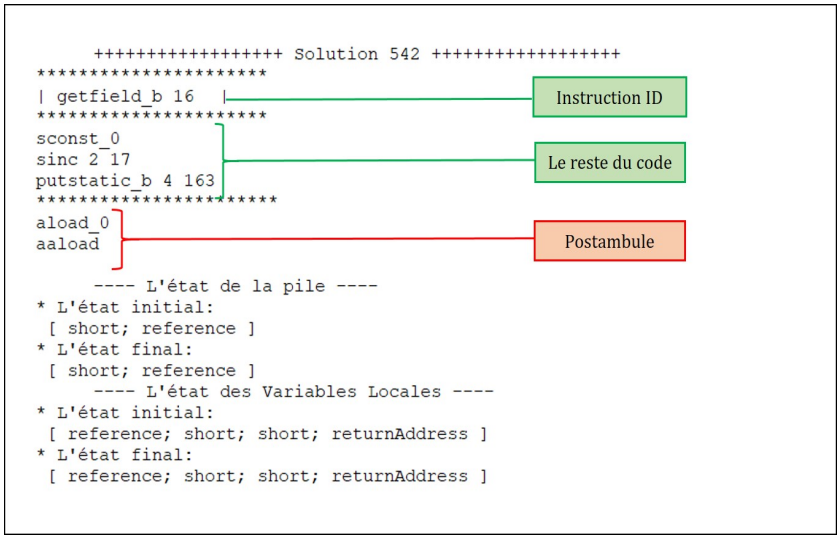

Fig 6.12 – Exemple de solution "avec postambule"

6.3 Etude de cas : appel des méthodes natives

L'objectif de l'étude de cas exposée dans cette section (présentée dans notre article [HLM19]) est double : nous voulons montrer l'utilité et l'efficacité de notre approche de construction de séquences de code (chapitre 4) tout en illustrant le mécanisme de désynchronisation (chapitre 5) à travers une exploitation possible.

Pour ce faire, nous avons choisi un cas pratique basé sur les résultats présentés par Mesbah et al. dans [MLM17]. Les auteurs ont montré comment ils ont pu localiser chaque classe, interface et

méthode de l'API Java Card. Par la suite, ils ont analysé et inverser le code de ces méthodes. Il a été constaté au cours de cette étape que certaines méthodes appelées utilisent des signatures spéciales différentes de celles définies dans la spécification de la machine virtuelle Java Card. En effet, il s'agit des en-têtes de méthodes natives. Sans avoir accès au code natif, la sémantique de méthodes appelées ainsi que leur conventions d'appels ont été inférées. De plus, les auteurs ont démontré la capacité d'appeler ces méthodes dans une application, au niveau Java, pour récupérer les données sensibles de la carte. Ceci en mettant en place une nouvelle attaque qui fournit un accès complet aux données cryptographique et permet de réinitialiser l'état de la carte à sa configuration initiale.

Notre but est de cacher les appels natifs trouvés dans [MLM17] en se basant sur les deux approches que nous avons proposé. L'appel de ces méthodes se fait à travers une instruction spéciale. Il s'agit de l'instruction `0xCD` qui nécessite deux octets utilisés comme token pour l'appel natif, ce qui nous donne `0xCD op1 op2`. Le tableau ci-dessous 6.1 résume les appels des méthodes natives trouvés et présentés dans [MLM17].

Appel natif (0x)	**Nom de la méthode**	**Appel natif (0x)**	**Nom de la méthode**
CD 21 80	readByteVMSTACK()	CD 22 42	writeByte()
CD 21 80	writeByteVMSTACK()	CD 33 42	writeShort()
CD 21 C0	readShortVMSTACK()	CD 73 42	xorify()
CD 22 C0	writeShortVMSTACK()	CD 62 42	deXorify()
CD 21 00	readByteRam()	CD B1 0F	deadCard()
CD 22 00	writeByteRam()	CD A3 1C	generateRandomData()
CD 21 40	readShortRam()	CD A1 1D	isAppletActive()
CD 22 40	writeShortRam()	CD 25 AB	encryption()
CD 22 02	readByte()	CD 25 2B	decryption()
CD 33 02	readShort()		

Tab 6.1 – Les appels aux méthodes natives

Afin de cacher l'appel d'une méthode native, nous avons appliqué notre principe de désynchronisation de code sur ce dernier. Suivant le modèle choisi dans le chapitre précédent, il faudrait insérer l'opcode d'une instruction bytecode juste avant cet appel (dans la suite, nous gardons ID pour instruction de désynchronisation). Nous avons choisi de considérer les instructions ayant besoin de deux opérandes afin de dissimuler la séquence. Cela provoquera la désynchronisation du code original. En d'autres termes, comme le montre la figure 6.13, les deux octets (`0xCD op1`) seront les opérandes de l'instruction ID et le deuxième opérande `op2` sera interprété comme une nouvelle instruction. Cependant, en plus du choix de l'instruction à ajouter (ID), la séquence de code la précédant (appelée préambule) doit être générée afin d'avoir l'état de la mémoire nécessaire à l'exécution correcte du code. L'approche proposée pour la construction de séquences de code est appliquée à cette étape. Le résultat final est un nouveau fragment de code où l'appel natif est caché tout en préservant la correction syntaxique et sémantique du code (satisfaction des contraintes).

Compte tenu de l'ensemble des appels natifs et de l'ensemble des instructions bytecodes avec deux opérandes, nous avons étudié les cas possibles à considérer. Nous avons procédé par élimination afin de réduire l'ensemble à traiter dans les expérimentations. Pour cela, nous sommes passés par les trois étapes suivantes :

1. Le choix des appels natifs à considérer

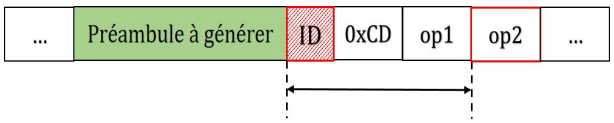

Fig 6.13 – Dissimulation d'un appel natif

2. Le choix de l'instruction ID
3. La génération de la séquence à ajouter (utilisation de l'outil *Trace Genarator*)

6.3.1 Etape 1 : Choix de l'appel natif à prendre en compte

Comme indiqué précédemment, l'octet `op2` est interprété en tant qu'une nouvelle instruction bytecode. En fonction de l'appel natif considéré, la valeur de cet octet va changer. Le tableau 6.2 résume toutes les valeurs possibles de cet octet ainsi que la pré-condition correspondant à chaque cas.

Cas	**Valeur de op2**	**Instruction**	**pré-condition**	**Appel Natif (0x)**
Cas 1	0xC0	Instruction réservée	Inconnue	CD 21 C0
				CD 22 C0
Cas 2	0x00	`nop`	none	CD 21 00
				CD 22 00
	0x02	`sconst_m1`	none	CD 22 02
				CD 33 02
	0x0F	`iconst_5`	none	CD B1 0F
	0x1C	`sload_0`	none	CD A3 1C
	0x1D	`sload_1`	none	CD A1 1D
Cas 3	0x80	`putstatic_b`	byte	CD 21 80
				CD 22 80
	0xAB	`getfield_s_w`	objectRef	CD 25 AB
	0x2B	`astore_0`	objectRef	CD 25 2B
Cas 4	0x40	`swap_x`	plusieurs octets	CD 21 40
				CD 22 40
	0x42	`iadd`	int int	CD 22 42
				CD 33 42
				CD 73 42
				CD 62 42

Tab 6.2 – Valeurs de l'octet op2 et leurs instructions correspondantes

Selon le tableau 6.2, quatre cas possibles sont distingués :

— *Cas 1 :* la valeur de `op2` est `0xC0` qui correspond à une instruction bytecode réservée selon la spécification de la machine virtuelle Java Card [Ora15b]. Ainsi, les appels natifs correspondants ne sont pas pris en considération.

— *Cas 2 :* la valeur de `op2` correspond à une instruction dont la pré-condition est vide (ne nécessite aucun élément). Une telle instruction n'est pas affecté par la post-condition de l'instruction ID. Ainsi, elle peut être exécutée sans problème.

— *Cas 3 :* la valeur de `op2` correspond à une instruction dont la pré-condition est égale à la post-condition de l'instruction ID. Une telle instruction peut être exécutée sans problème aussi.

— *Cas 4 :* la valeur de `op2` correspond à une instruction dont la pré-condition nécessite un ou

plusieurs éléments qui ne sont pas compatibles avec la post-condition de l'instruction ID. De telles instructions nécessitent plus d'investigation.

6.3.2 Etape 2 : Choix de l'instruction ID

La spécification de la machine virtuelle Java Card [Ora15b] définit 43 instructions bytecode ayant deux opérandes. Après l'étude des valeurs possibles pour l'instruction ID (en se basant sur je jeu d'instructions actuellement pris en charge par l'outil *Trace Generator* ainsi que les valeurs réalistes des différents opérandes), nous avons conservé les 10 instructions suivantes : `anewarray, getstatic_a, getstatic_b, getstatic_s, jsr, new, putstatic_a, putstatic_b, putstatic_s, sspush`. Le tableau 6.3 liste les instructions considérées ainsi que leurs pré-conditions et post-conditions.

Instruction ID	Opcode (0x)	Pré-condition	Post-condition
`anewarray`	91	short	arrayref
`getstatic_<t>`	7B .. 7E	/	short, objectRef
`jsr`	71	/	address
`new`	8F	/	objectRef
`putstatic_<t>`	7F..82	`value_<t>`	/
`sspush`	11	/	short

Tab 6.3 – Les valeurs possibles de l'instruction ID

6.3.3 Etape 3 : Génération de la séquence à rajouter

Dans cette étape de la génération des solutions, pour chaque appel natif considéré dans étape 6.3.1, nous prenons toutes les valeurs possibles de l'instruction ID (étape 6.3.2) afin de générer les solutions possibles à l'aide de l'outil *Trace Generator*. En fonction des deux paramètres : la taille maximale de la séquence à générer (elle représente la profondeur de l'arbre de recherche) et le nombre maximum de solutions (qui varie de 100 à 1 million) ; le nombre de solutions trouvées et le temps de génération qui est nécessaire sont enregistrés dans le tableau 6.4.

Pour ce faire, nous avons considéré les entrées suivantes pour l'outil :
- Instruction de départ : l'instruction ID (10 choix possibles)
- Pile des opérandes de départ : pré-condition de l'instruction ID considérée
- Pile des opérandes d'arrivée : pile vide
- Liste des variables locales : `reference, shortn reference` (il s'agit d'un exemple)
- Mode de génération : classique

Nous avons constaté que l'instruction ID ne dépend pas de la valeur des octets de l'appel natif (ses 2 opérandes). Donc, quel que soit l'appel natif considéré, nous aurons la même expérimentation. Pour cela les résultats de la génération des solutions sont représentés en un seul tableau (6.4) indépendamment de l'appel natif considéré. Les expériences sont réalisées sur un ordinateur personnel doté d'un processeur i7-6500u 3.1Ghz.

A travers ces résultats, nous remarquons que les instructions ID considérées peuvent être classées en deux catégories d'instructions :
- *Catégorie 1* : Concerne les instructions présentées dans le tableau 6.4. Il s'agit des instructions `getstatic_a, getstatic_s, putstatic_a, sspush`. Nous pouvons avoir un nombre de solutions qui dépasse 1000 en quelques secondes et plus d'un million en quelques de minutes (entre 7 et 14,5 minutes). Ce qui est un délai très raisonnable pour un tel nombre de solutions

solution size	$getstatic - a$		$getstatic - s$		$putstatic - a$		$sspush$	
	# sol*	time ms	# sol	time ms	# sol	time ms	# sol	time ms
1	3	4	2	3	1	3	4	3
2	5	6	4	4	2	3	20	9
3	23	8	15	6	2	3	91	26
4	47	24	31	14	3	10	289	88
5	178	55	99	33	16	15	1045	294
6	581	200	286	126	27	53	3423	1123
7	2123	743	1140	403	98	210	13378	4489
8	7663	3038	3746	1698	516	350	49313	17909
9	31833	11886	16001	6880	1651	1393	200379	71205
10	117080	48117	54889	27070	6275	5736	733922	288127
11	464974	219854	231265	109069	25148	23613	>1000000	410241
12	>1000000	510000	844996	511202	108213	105199		
13			>1000000	860946	425547	440450		
14					>1000000	448738		

Tab 6.4 – Résultats expérimentaux obtenus par l'outil Trace generator

* nombre de solutions générées

différentes. Les solutions trouvées doivent faire l'objet d'une phase de vérification (section 4.5.4) afin qu'elles puissent être chargées dans une carte réelle.

— *Catégorie 2* : Concerne les instructions qui ne sont pas présentées dans le tableau 6.4 et pour lesquelles une solution unique est générée quelle que soit la taille maximale choisie pour la séquence à générer. Cela est dû au fait que :

  — dans le fichier statistiques des bigrams, la ligne correspondant à l'instruction ID est vide (c'est-à-dire qu'il n'y a pas de données). Autrement dit, le nœud racine (les instructions ID considérées) n'a pas d'instructions candidates pouvant le précéder. C'est le cas des instructions ID : `putstatic_b,putstatic_s, jsr`. Ce cas pourrait être traité par l'enrichissement du fichier des statistiques.

  — les instructions candidates de l'instruction ID ne trouvent pas leurs post-conditions au sommet de la pile, i.e. la post-condition de l'instruction candidate est différente de la pré-condition de l'instruction ID. C'est le cas des instructions : `new, getstatic_b`. Ce cas pourrait être étudié plus en détails afin de générer plus de solutions. Par exemple, ceci est possible en trouvant l'état mémoire initial approprié (i.e. compatible avec la post-condition de l'instruction candidate). Mais l'effet sur le postambule du code (i.e. le code qui vient après l'appel natif) reste à vérifier, aussi.

6.3.4 Exemples d'appels natifs cachés

Afin d'illustrer les résultats obtenus, nous présentons deux exemples de dissimulation d'un appel natif en se basant sur les explications données dans les sections ci-dessus. Toutefois, d'autres exemples de combinaisons possibles (appel natif, instruction ID) peuvent être réalisés en suivant les mêmes étapes. Auparavant, selon les tableaux 6.2 et 6.3, les appels natifs considérés (cas 2 et 3 de la section 6.3.1) ainsi que leurs instructions ID correspondantes sont résumés dans le tableau 6.5.

Valeur op2	Instruction	Pré-condition	Appel natif (0x)	Instruction ID
0x00	`nop`	none	CD 21 00	
			CD 22 00	Toutes les instructions ID considérées dans la section 6.3.2
0x02	`sconst_m1`	none	CD 22 02	
			CD 33 02	
0x0F	`iconst_5`	none	CD B1 0F	
0x1C	`sload_0`	none	CD A3 1C	
0x1D	`sload_1`	none	CD A1 1D	
0x80	`putstatic_b`	byte	CD 21 80	`getstatic_b`,
			CD 22 80	`getstatic_s`,
				`sspush`
0xAB	`getfield_s_w`	objectRef	CD 25 AB	`new, getstatic_a`
0x2B	`astore_0`	objectRef	CD 25 2B	`new, getstatic_a`

Tab 6.5 – Les appels natifs considérés ainsi que les instructions ID correspondantes

6.3.4.1 Exemple 1 : la pré-condition de l'instruction op2 est vide

C'est un exemple du cas 2 (section 6.3.1). Le but est de cacher l'appel de la méthode native `readByteRam()` i.e. il est considéré comme étant le code hostile à dissimuler. Pour cela, nous avons choisi l'instruction `putstatic_a` comme étant l'instruction ID à rajouter avant l'appel natif. Le code obtenu est présenté à la figure 6.14. L'instruction `putstatic_a` prend les deux premiers octets de l'appel natif comme étant ses opérandes (`0xCD 0x21`). Ainsi, l'octet restant (la valeur de l'octet `op2` est `0x00`) est interprété comme une instruction `nop`.

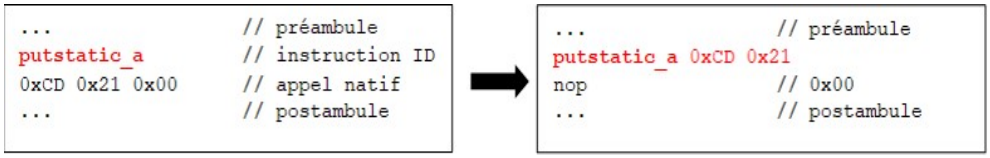

Fig 6.14 – Cacher l'appel de la méthode native readByteRam()

Il reste à générer la séquence de préambule correspondant à la pré-condition de instruction ID (i.e `putstatic_a`. L'outil *Trace Generator* fournit un ensemble de solutions possibles. Parmi lesquelles celle présentée dans la figure 6.15 accompagnée de son exécution (les différents états de la pile des opérandes).

6.3.4.2 Exemple 2 : instructions op2 avec une pré-condition

C'est un exemple du cas 3 (section 6.3.1). Rappelons que dans ce cas la pré-condition de l'instruction `op2` correspond à la post-condition de l'instruction ID. Le but dans cet exemple est de cacher l'appel de la méthode native `decryption()` i.e. il est considéré comme étant le code hostile à dissimuler. Pour cela, nous avons choisi l'instruction `getstatic_a` comme étant l'instruction ID à rajouter avant l'appel natif. Le code obtenu est présenté à la figure 6.16. L'instruction `getstatic_a` prend les deux premiers octets de l'appel natif comme étant ses opérandes (`0xCD 0x25`). Ainsi, l'octet restant (la valeur de l'octet `op2` est `0x2B`) est interprété comme une instruction `astore_0`.

Parmi les différentes solutions possibles générées par l'outil *Trace Generator*, celle présentée dans la figure 6.17) accompagnée de son exécution (les différents états de la pile des opérandes).

Quel que soit le cas traité, si après l'injection de la faute (l'octet cible est l'instruction ID), il y

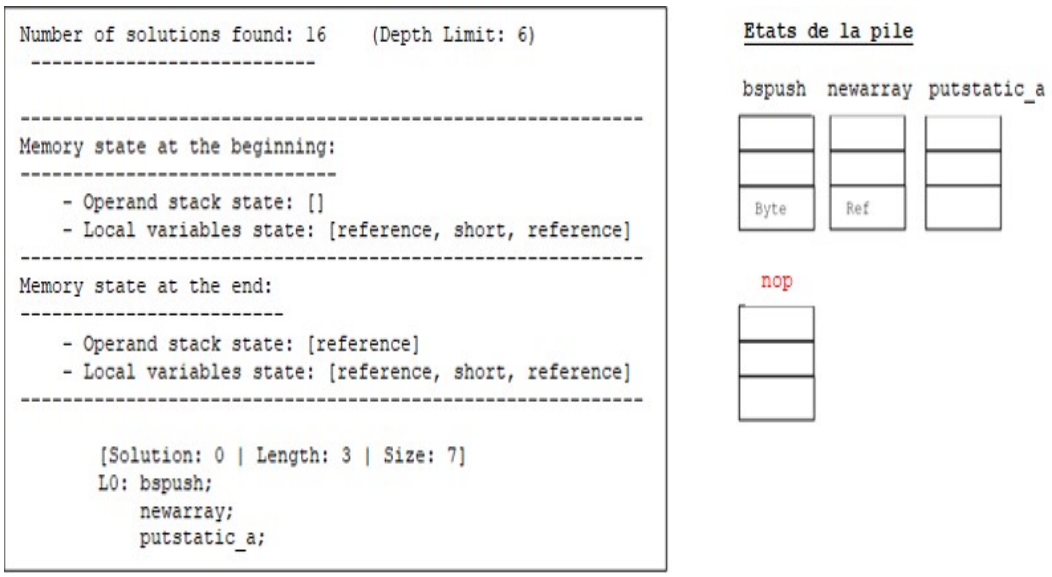

Fig 6.15 – Exemple d'une solution générée par l'outil Trace Generator (1)

Fig 6.16 – Cacher l'appel de la méthode native decryption()

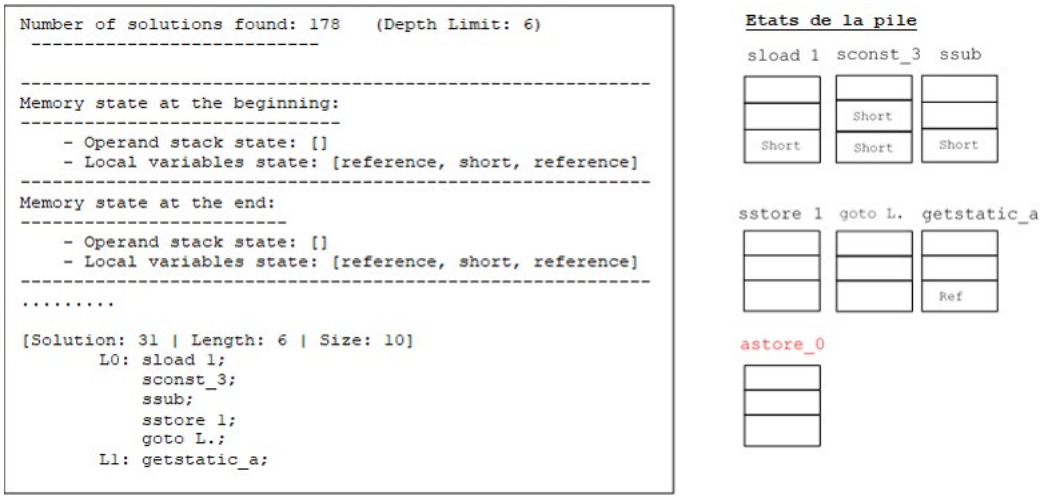

Fig 6.17 – Exemple d'une solution générée par l'outil Trace Generator (2)

a encore des éléments restants sur la pile (générés par le préambule), ils doivent être gérés dans un postambule (afin de les consommer). Cette opération est effectuée pour récupérer le flot d'exécution original.

6.4 Détection des mutants : analyse de vulnérabilités à l'injection de fautes

Avec l'émergence des attaques combinées, le besoin de l'évaluation de la robustesse des systèmes faces aux injection de fautes (moyen d'activation de ces attaques) ne cesse d'augmenter. Ceci dans le but de s'assurer du niveau de sécurité offert par le système et qui est à la base de l'instauration de confiance avec ses utilisateurs finaux. Ceci a donné lieu à plusieurs travaux ayant comme objectif commun la reproduction de l'effet des fautes pour la détection des vulnérabilités dues aux injections de fautes. Il existe deux grandes catégories d'approches [GWJL18] :

— *Les approches basées sur le logiciel (Software based approaches)*. Elles consistent à effectuer une simulation de l'injection de fautes, basée sur un modèle de fautes choisi, telle qu'elle puisse se produire lors d'une attaque réelle. Ceci, afin de pouvoir évaluer son effet sur le système et ainsi détecter d'éventuelles vulnérabilités qui pourraient être exploitées. De telles approches sont peu coûteuses et rapides à mettre en œuvre puisqu'elles nécessitent seulement des compétences de développement sans avoir besoin d'un matériel spécialisé.

— *Les approches basées sur le matériel (Hardware based approaches)*. Elles consistent à utiliser des équipements spécialisés pour injecter une faute réelle sur un dispositif spécifique. Ainsi, toute vulnérabilité détectée est garantie d'être réelle et potentiellement reproductible. Cependant, elles nécessitent un matériel spécialisé dont le coût est élevé ainsi que de l'expertise pour configurer un tel environnement et réaliser les expériences.

Actuellement, la plupart des environnements d'injection de fautes proposés pour les cartes à puce sont basés sur l'injection de fautes logicielles [LBH+14] (i.e. appartiennent à la classe des *approches basées sur le logiciel*). Comme il s'agit de notre domaine d'étude et que notre travail s'inscrit dans la catégorie des attaques combinées, notre intérêt porte sur les outils de simulation des injections de fautes dans le but d'étudier la possibilité de détection de nos mutants (i.e. les code hostiles activables par attaque en faute).

Ceci dit, le but de cette section est de présenter quelques outils d'analyse existants, leurs limites ainsi que des pistes de solutions qui pourront être envisagées dans des travaux futurs.

6.4.1 Les outils de simulation des injection de fautes

Un état de l'art assez intéressant sur les outils de simulation des injections de fautes est présenté par Dureuil dans [Dur16]. Il présente une panoplie d'outils existants ainsi qu'une classification selon des critères jugés pertinents. Ces derniers, sont résumés comme suit :

— *Approche de détection* : domaine applicatif, niveau de l'analyse (modèle, source, assembleur, binaire), niveau d'exécution du code (embarqué sur la carte, simulation dynamique, exécution symbolique, exécution concolique).

— *Modèle de faute* : perturbation du flot de données, perturbation du flot de contrôle, remplacement d'instruction.

— *Campagnes d'injection* : gestion des entrées utilisateur, état initial de l'exécution, liste des fautes à injecter, gestion des fautes multiples.

— *Méthode de traitement des résultats* : manuel, par comportement, par métriques d'évaluation, par preuve.

Ci-dessous, on va reprendre les outils qui reviennent le plus souvent dans la littérature (la liste n'est pas exhaustive) et dont le domaine applicatif est les cartes à puce (mais pas spécialement Java Card).

Outil de simulation d'attaque en SystemC

Rothbart et al. [RNS+04] ont proposé un outil d'injection de fautes au niveau des modèles de SystemC [3]. L'outil fonctionne en ajoutant des blocs spéciaux d'injection de fautes entre les différents modules du modèle fonctionnel et les mémoires. Ces blocs interceptent les requêtes d'accès mémoire et renvoient des résultats fautés. Ces derniers sont ensuite analysés en comparant les sorties des exécutions fautées avec les sorties nominales.

Outil SmartCM

Machemie et al. [MMLC11] ont proposé un outil qui simule l'injection de fautes dans des codes Java Card en mutant le bytecode Java. Il est basé sur la stratégie de mutation et modifie les données et le code. L'outil génère exhaustivement tous les mutants possibles (Force brute). A chaque mutation, il y a remplacement de la valeur d'un octet du bytecode. Par la suite, l'outil est capable d'éliminer automatiquement les mutants qui sont mal formés (ne correspondent pas à un programme Java correct). Les mutants restants sont évalués pour décider s'ils sont dangereux ou non.

Outil de Kauffmann-Tourkestansky

Dans sa thèse, Kauffmann-Tourkestansky [KT12] a proposé une formalisation des propriétés de sécurité ainsi qu'un modèle d'attaque au niveau code source. Il s'intéresse spécialement aux effets des fautes sur le flot de contrôle du programme. L'objectif de son outil est de permettre à un développeur qui écrit le code à haut niveau (typiquement, en langage C) de tester la présence de vulnérabilités dues à l'injection de fautes à bas niveau (assembleur). L'approche retenue est la mutation de programmes C suivant un double modèle d'assignation des variables et de saut exploitant l'instruction `goto`.

Outil EFS (Embedded Fault Injection Simulator)

Berthier et al. [BBC+14] ont proposé EFS, un simulateur d'injection de fautes embarqué sur carte. Pour cela, ils ont intégré un mécanisme d'injection de fautes directement dans la carte à puce fournissant des commandes de contrôle pour configurer les attaques et récupérer les données pertinentes. Cela permet d'avoir accès aux traces de consommation de courant correspondantes à chaque attaque effectuée, et qui sont par la suite analysées via des observations par canaux cachés. EFS utilise les mécanismes d'interruptions programmables pour simuler l'injection de fautes suivant un modèle de faute de sauts d'instructions. L'outil ne peut être embarqué dans la carte qu'après la production du premier prototype du circuit. Donc tout changement à ce niveau avancé du projet est extrêmement coûteux.

Outil de Chorko et al.

Chorko et al. [CKN14] ont proposé un autre environnement d'injection de fautes Java Card. Ils ont utilisé le fichier binaire, librement disponible, du simulateur *cref* de Java Card. Il s'exécute sur un ordinateur de bureau, comme un environnement Java Card. Ils ont écrit un plugin pour *cref*. Son objectif est d'accéder aux données et de les manipuler à l'intérieur de la mémoire non volatile simulée (elle contient le segment bytecode). Le plugin est capable de modifier l'instruction courante, ses paramètres et le contenu de la mémoire en fonction de la méthode sélectionnée. Cependant, cet outil ne peut pas injecter des fautes dans la mémoire principale qui est utilisée par la machine virtuelle.

3. SystemC est une bibliothèque C++ qui permet de modéliser des systèmes au niveau comportemental. Il est souvent comparé aux langages Verilog et VHDL.

Les auteurs indiquent une couverture complète du code avec un paramètre de perturbation généré aléatoirement. Ce qui correspond à une approche par force brute (semblable à SmartCM).

Outil CELTIC

Dureuil et al. [DPdC+15] ont proposé un simulateur dynamique de code binaire, capable d'injecter des fautes durant la simulation du code. CELTIC exécute une exécution de référence puis, sur la même trace, implémente toutes les instances de fautes possibles selon le modèle de faute choisi. Le traitement des résultats utilise un oracle fourni en entrée. Ceci dit, après chaque exécution fautée l'état du processeur simulé est inspecté pour déterminer s'il vérifie un prédicat donné. Ceci dans le but de prendre une décision concernant le succès ou l'échec de l'injection de faute.

Classifieur de mutants

Yahiaoui et al. [YLMT17] ont proposé un classifieur de mutants pour prédire les patterns dangereux dans des applications Java Card. L'approche proposée est basée essentiellement sur des techniques de reconnaissance de texte et de "machine learning". Les premières sont utilisées afin de fournir au classifieur des patterns dans le format adéquat pour effectuer les traitements de la phase suivante. Cette dernière est basée sur un schéma de classification supervisée dont le processus global est divisé en deux étapes : l'apprentissage et le test. Dans la phase d'apprentissage, un ensemble d'objets sont classifiés par un expert (dangereux ou non). Chaque objet est représenté par un vecteur de caractéristiques. Puis, ces objets sont utilisés pour faire apprendre le classifieur à travers un algorithme de "machine learning" approprié. Par la suite, durant la phase de test, le classifieur précédent est testé à l'aide d'un ensemble d'objets (d'autres objets ne figurant pas dans l'ensemble initial) afin d'évaluer son exactitude. Les résultats expérimentaux ont montré une amélioration du temps d'exécution par le classifieur par rapport à l'outil SmartCM.

6.4.2 Limites et pistes de solutions

En se basant sur un modèle de faute précis, la majorité des outils de simulation existants génèrent les fautes possibles de façon exhaustive afin d'avoir une couverture totale des fautes. Autrement dit, ceci a pour avantage de ne pas laisser de côté un cas qui pourrait être problématique. Cependant, l'inconvénient majeur d'une telle approche est l'explosion combinatoire de l'espace des fautes à traiter. En plus du nombre important des injections de fautes à réaliser, la majorité du temps ces dernières sont sans effet sur le comportement du code (mutants inutiles) [BGL19].

Ceci dit, il y a besoin d'introduire de l'intelligence dans la génération des mutants. Plusieurs travaux qui visent à réduire le nombre des expérimentations à réaliser (i.e. les injections de fautes) ont vu le jour. Leurs origines remontent au domaine, plus général, de la réduction des suites de test. Ils correspondent à des techniques d'optimisation allant du simple *pruning* aux techniques plus élaborées de ce dernier. Dans ce qui suit, nous présentons brièvement les techniques les plus répandues dans la littérature dans l'optique de les adapter à la problématique de détection de nos mutants (conçus selon notre approche présentée dans cette thèse). Ceci dans le but de mettre en place par la suite les outils associés dans nos travaux futurs : il s'agit des pistes à explorer.
Pour plus de détails sur ces techniques ainsi que d'autres, les références [Böc12, HANR12, SBS15, BGL19] peuvent être consultées. Ces travaux sont issus de plusieurs domaines d'applications mais aucun ne traite le cas spécifique des applications Java Card.

6.4.2.1 Le *pruning* simple (fault simpling)

Cette technique consiste à échantilloner l'espace des fautes en se basant sur la supposition que la distribution des fautes dans cet espace est uniforme [BRIM98]. Ceci résulte en la sélection d'un sous-ensemble de fautes pour lequel les injections de fautes vont être réalisées. Ce sous-ensemble est considéré comme étant représentatif de l'ensemble complet des fautes. Bien que cette technique soit rapide et simple à implémenter, mais clairement elle ne couvre pas toutes les fautes possibles donc il y a possibilité de passer à côté des effets des fautes qui n'ont pas été sélectionnées (problème de sous-estimation). Ceci dit, les éventuelles vulnérabilités associées à ces dernières ne seront pas détectées.

6.4.2.2 Quelques techniques du *pruning* heuristique

Limitation des adresses

Dans certains cas, les injections de fautes peuvent faire que les applications accèdent à des emplacements mémoire qui se situent hors de la plage de l'espace d'adressage alloué. De tels accès sont susceptibles d'aboutir à des problèmes détectables (erreurs de segmentation, interruption de l'application, *etc*). Donc, il n'y a pas besoin d'injecter de telles fautes afin d'analyser leur effet car ce dernier est connu à l'avance. Ceci dit, elles peuvent être éliminées de l'espace des fautes au début.
Cette technique de limitation des adresses est une technique de *pruning* à résultat connu (*Known-outcome pruning*). L'idée est de déterminer la plage d'adresses valides (sur un "*golden run*"[4]), à la fois pour la pile et pour le tas. A l'aide de ces informations, toutes les fautes qui produisent une nouvelle adresse n'appartenant pas à cette plage sont considérées comme étant des fautes détectables. Ainsi, il n'y aura pas d'injection pour les fautes de cette dernière catégorie : elles sont éliminées.
Nous pourrons appliquer cette technique sur les instructions d'accès à la mémoire ("de" et "vers" la pile des opérandes et la liste des variables locales). Et même l'entendre aux instructions de branchement manipulant des adresses mémoire. Les fautes produisant des adresses invalides sont éliminées (i.e. les mutants associés).

Analyse *def/use* conservatrice

Les techniques les plus sophistiquées du *pruning* sont basées sur les traces d'instructions et d'accès à la mémoire. Ces traces sont créés au cours d'un "*golden run*", qui exécute le programme cible sans injecter des fautes [SBS15].
Basé sur ce type d'informations sur les traces d'accès en mémoire, Smith et al. [SJPB95] sont parmi les premiers à décrire la technique classique d'analyse de la *def/use*. Cette technique a été reprise plusieurs fois par la suite dans d'autres travaux de recherche (nous citons Benso et al [BRIM98], Berrojo et al. [BGC+02], Barbosa et al. [BVFK05], Grinschgl [GKS+12], Hari et al. [HANR12] et dernièrement Boespflug et al. [BGL19]).
Cette méthode réduit de façon conservatrice l'espace de fautes i.e. sans compromettre la qualité des résultats. L'idée de base est que tous les emplacements de fautes entre un *def* (une *écriture*) ou un *use* (une *lecture*) des données en mémoire, et une *lecture* suivante, sont équivalentes. Ceci indépendamment de l'emplacement de l'injection de faute dans cet intervalle car la donnée erronée ne serait visible au moment de la *lecture*. De plus, si l'injection de faute engendre une opération d'*écriture* qui sera suivie par une autre *écriture* dans le programme avant la *lecture*, la faute en question est sans effet. De même, toutes les injections de faute dans un intervalle de temps entre une *lecture* ou *écriture* et une *écriture* suivante seront sans effet car la donnée erronée serait écrasée dans tous les cas. Ceci dit, au lieu d'avoir une injection de faute pour chaque point dans cet intervalle de temps, il suffit de réaliser une seule expérience (par exemple, juste avant la *lecture*), en considérant que c'est le même résultat qui sera

4. C'est une exécution sans erreurs (fault-free) qui sert de référence pour le comportement attendu du programme

obtenu pour les autres points de cet intervalle.
Donc, au final l'espace de fautes est partitionné en classes d'équivalence *def/use*. Pour chaque classe, une seule expérience doit être menée. Les autres fautes de la même classe ont un résultat d'expérience a priori connu. En faisant une projection sur notre domaine d'application, ce sont les mutants qui appartiennent à cette dernière catégorie qui doivent être éliminés afin de réduire le nombre de fautes à injecter.

Equivalence du flot de contrôle

Cette technique est beaucoup plus sophistiquée et repose sur les observations suivantes [HANR12] :

— Des fautes similaires se propagent à travers des séquences de codes similaires.

— Les exécutions multiples d'une séquence de code augmentent la probabilité d'erreurs.

L'idée de cette technique est similaire à celle de la technique précédente. Soit une instruction statique I ayant plusieurs instances dynamiques I_d. Ces instances dynamiques sont divisées en classes d'équivalence en se basant sur le chemin du flot de contrôle suivi après une instance dynamique. L'injection de faute est réalisée uniquement sur un représentant de chaque classe, appelé *pilote*. La difficulté avec cette technique réside dans la définition de l'équivalence qui utilise les observations données ci-dessus.
La technique opère au niveau bloc de base. Une exécution *"golden run"* est utilisée pour énumérer tous les chemins possibles du flot de contrôle. Ceci en commençant par le bloc de base qui contient l'instruction cible et allant jusqu'à une profondeur n. La profondeur est définie comme étant le nombre d'instructions de branchement ou de saut rencontrées. Pour les chemins qui ont été parcourus plusieurs fois durant l'exécution, une instance dynamique *pilote* est sélectionnée au hasard. Toutes les autres exécutions non sélectionnées pour ces chemins sont éliminées, en supposant que les fautes visant ces exécutions sont représentées par l'instance dynamique pilote.

Nous pouvons conclure que les techniques du pruning constituent une piste assez puissante et efficace pour régler le problème de réduction de la taille de l'espace des fautes par l'élimination de celles qui sont inutiles. C'est une piste prometteuse sur laquelle va porter nos futures recherches en étudiant en détails les différentes techniques existantes dans le but de déterminer celles qui pourraient être adaptées ou combinées pour mettre en œuvre des outils de détection de mutants, spécifiques Java Card, dotés d'une certaine intelligence.

6.5 Conclusion

Dans ce chapitre, nous avons pu montrer comment nos deux approches (combinées ou séparées) nous permettent de répondre à la problématique principale soulevée au début de ce travail :« Comment construire un code malveillant activable par attaque en faute ? ». Ceci, en présentant des exemples d'exploitations possibles utilisant les outils développés.
On ne s'est pas intéressé à donner une évaluation quantitative des solutions pouvant être générées car ce n'est pas ce qui importe dans notre cas. Mais plutôt, il suffit de trouver une seule solution qui répond aux critères pour affirmer qu'il est possible de cacher un code dans un autre en procédant par construction. De plus, les résultats retournés par les outils dépendent essentiellement des entrées qui leur sont fournis.

Conclusion et Perspectives

Dans cette thèse, nous avons présenté un nouveau risque pouvant compromettre la sécurité d'une carte à puce. Ce vecteur d'attaque s'inscrit dans la catégorie des attaques combinées qui ont pris de l'ampleur dans les dernières recherches car elles bénéficient à la fois des points fort des attaques logicielles et ceux des attaques matérielles.

La question de départ était relative à la possibilité de concevoir une application légitime qui peut passer toutes les vérifications. Une fois sur la carte, elle subit une mutation intentionnelle moyennant une injection de faute ciblée afin d'activer un comportement hostile caché. Au début, nous ne savions pas si cette idée était « un mythe ou une réalité ». La réponse est venue suite à une preuve de concept qui nous a permis de montrer que bien que cette idée soit assez difficile à mettre en œuvre mais elle n'est pas impossible, donc c'est bien une réalité qu'il faudrait étudier en détails afin d'aboutir à la solution attendue.

Au départ, nous avons trouvé des fondements théoriques à notre problème en montrant qu'il se ramène à un problème de satisfaction des contraintes. En effet, nous essayons de dissimuler un code hostile dans un code inoffensif par rajout de quelques instructions avant son début de telle façon à ce que la séquence constituant le code hostile n'apparaisse plus sous sa forme originale. Le code rajouté est une séquence d'instructions choisies parmi celles définies dans la spécification de la machine virtuelle Java Card tout en respectant un ensemble de contraintes. Il s'agit de s'assurer que le le code résultant est syntaxiquement et sémantiquement correct vis-à-vis de cette spécification. De plus, ce rajout de code a un effet sur l'interprétation du reste du programme que nous avons appelé désynchronisation de code. Il s'agit d'un décalage induit par le changement de la séquence hostile initiale suite au rajout des instructions avant son début. Ceci dit, il fallait trouver deux solutions pour les deux problèmes soulevés afin de dissimuler un code : la construction de la séquence de code et la désynchronisation de ce dernier.

Pour le premier problème, nous avons proposé une approche de construction de code basée sur la satisfaction des contraintes et un algorithme de parcours d'arbre. Le raisonnement adopté est dans le sens inverse de l'exécution normale d'un code donné. En effet, partant du code hostile nous visons à construire une séquence de code pouvant le précéder : construction en parcours arrière. Le problème à été représenté sous la forme d'un arbre de recherche qui est généré et exploré à la volet. Tous les chemins menant de la racine aux feuilles de l'arbre constituent des solutions possibles au problème i.e. des séquences de code pouvant être rajoutées avant le code hostile. Afin d'améliorer la qualité des solutions générées, nous avons proposé une optimisation basée sur des heuristiques qui ont été définies (bi-grams et tri-grams). L'idée est de pondérer les instructions de telle sorte qu'elles soient ordonnées et explorées en fonction de leurs priorités.

Pour le second problème, nous avons commencé par définir ce qu'un modèle de désynchronisation. Par la suite, pour un modèle choisi, nous avons défini que représente un code correctement désynchronisé. Sur la base de ces concepts et suite à une étude détaillée du problème, nous avons proposé une approche appliquant le mécanisme de désynchronisation sur un code donné dans le but de le dissimuler. Elle traite le cas général ainsi que tous les cas particuliers qui en découlent. Pour traiter ces derniers, cette approche se base en partie sur la première approche de construction de séquences de code.

La mise en pratique des deux approches a été faite par l'implémentation de deux outils correspondants. L'objectif de chacun est d'automatiser une approche. Le premier pour générer toutes les séquences de code possibles liant deux fragments de code donnés. Et le second, pour calculer tous les décalages possibles d'un code donné. Afin de confirmer la faisabilité des deux approches proposées, montrer l'efficacité de ces deux outils et donner des exploitations possibles de notre travail, des exemples d'application et une étude de cas ont été présentés.

A l'issue de cette thèse, plusieurs perspectives de recherche apparaissent dans le but d'améliorer nos contributions et leur donner une suite dans le cadre de travaux futurs. Les différentes pistes d'extension possibles peuvent être résumées comme suit :

- **Généricité des approches**. Notre travail a été appliqué sur un exemple d'éléments sécurisés qui est la carte à puce et pour une plateforme cible qui est Java Card. Il serait intéressant d'étudier la possibilité de l'étendre à d'autres supports d'exécution manipulant d'autres langages. A priori, cette piste semble envisageable en gardant les principales idées des deux approches avec une adaptation aux spécificités du langage cible. Autrement dit, il faudrait revoir la modélisation du problème en fonction de ce dernier (le jeu d'instructions, les contraintes).
- **Construction de séquences de code**. Dans le but d'améliorer encore l'approche proposée, deux éléments pourraient être optimisés. Le premier, concerne le mécanisme du *backtracking* dans l'arbre de recherche. Au lieu de faire un simple retour arrière (i.e. un *backtraking chronologique*) si un critère d'arrêt est détecté, nous pouvons étudier la possibilité d'utiliser un mécanisme de *backtraking intelligent*. Le second point concerne la mise en place de nouvelles heuristiques utilisées pour le choix de l'ordre d'exploration des nœuds fils dans l'arbre. Nous pensons à intégrer la profondeur de l'instruction dans l'arbre pour les tri-grammes. Par exemple, ça pourrait nous éviter de sélectionner au début de la séquence construite des instructions qui ne peuvent pas y figurer dans des applications réelles.
- **Désynchronisation de code**. Comme expliqué précédemment, nous avons défini des paramètres qui représentent les modèles de désynchronisation possibles. Dans cette thèse, notre étude a porté sur un modèle spécifique parmi ceux qui peuvent être choisis. Il serait intéressant d'étudier les autres modèles qui se dégagent à partir d'autres combinaisons possibles des paramètres. Et apporter les traitements nécessaires aux nouveaux cas problématiques qui peuvent surgir.
- **Aspects techniques : outillage**. D'abord il faudrait compléter le jeu d'instructions manipulé par les deux outils par les instructions non prises en compte dans les versions actuelles. Par la suite, il serait bénéfique d'avoir une chaîne d'outils complète qui comprend toutes les étapes par lesquelles passe un code à cacher et va jusqu'à la phase de manipulation et vérification du fichier CAP à charger dans la carte.
- **Exploitations du vecteur d'attaque**. Nous avons présenté quelques exploitations possibles de nos approches. Et pour consolider encore le présent travail, il serait intéressant de revoir les attaques existantes pour étudier la possibilité de les rejouer dans le but de relaxer les contraintes

correspondantes (par exemple la présence d'un vérifieur de bytecode) et donc avoir des attaques plus puissantes.

— **Passage à l'ordre supérieur : un modèle multi-fautes**. Le modèle de faute adopté dans cette thèse, pour injecter la faute activant le code hostile, est un modèle mono-faute où la cible est un octet précis qui prendra la valeur `0x00`. Il y a eu des travaux de recherche qui ont portés sur les modèles multi-fautes (ou dits, fautes multiples) où un attaquant à la possibilité d'effectuer plusieurs perturbations différentes durant une seule exécution de son code ce qui augmente considérablement les pouvoirs de l'attaquant [Dur16]. C'est une piste à explorer pour voir la possibilité de ce qui pourrait être apporté dans notre cas. Autrement dit, est-ce que le fait de pourvoir injecter plusieurs fautes à la fois pourrait nous simplifier la conception de nos codes malveillants, ou bien ça rajoute encore de la complexité au travail ?

— **Volet contre-mesures**. Dans le cadre de cette thèse nous nous sommes focalisé sur l'étude et la contribution dans le volet « attaque ». Cet objectif étant atteint, nous disposons de la base nécessaire pour aborder le second volet qui est la « défense ». Il faudrait travailler dans ce sens pour mettre en place de nouvelles contre-mesures bien adaptées à ce que nous savons concevoir. Dans la section 6.4, nous avons présenté une piste prometteuse qui nécessite encore plus d'investigation pour pouvoir mettre en place des outils de détection appropriés. En outre, nous pouvons clôturer par la question suivante : « Pourrait-on aussi exploiter la piste des méthodes formelles pour la détection de nos codes malveillants activables par attaques en faute ? »

Troisième partie

Annexes

ANNEXE A

Classification des instructions bytecode selon le type d'opération

Manipulation de la pile des opérandes et la liste des variables locales (64) :

- **Empiler une constante dans la pile des opérandes (20)**
  - byte : bpush, bipush
  - short : sconst_<s>, sspush, sipush
  - int : iconst<i>, iipush
  - ref : aconst_null
  - note : $s, i \in \{m1, 0, 1, 2, 3, 4, 5\}$

  ****
- **Charger une variable locale dans la pile des opérandes (19)**
  - byte : /
  - short : sload, sload_<n>
  - int : iload, iload_<n
  - ref : aload, aload_<n>
  - array : aaload, baload, saload, ialload
  - note : $n \in \{0, 1, 2, 3\}$

  ****
- **Sauvegarder un élément de la pile des opérandes dans la liste des variables locales (19)**
  - byte :
  - short : sstore, sstore_<n>
  - int : istore, istore_<n>
  - ref : aastore, bastore, sastore, iastore
  - array : astore, astore_<n>
  - note : $n \in \{0, 1, 2, 3\}$

  ****
- **Instructions non-typées qui modifient la pile des opérandes (6)**
  - un élément : pop, dup
  - deux élément : pop2, dup2
  - plusieurs éléments :dup_x, swap_x

****

Instructions de conversion de type (4)

- short : s2b, s2i
- int : i2b, i2s

****

Instructions des opérations arithmétiques (16)

- short : saad, ssub, smul, sdiv, srem, sneg, sinc, sinc_w
- int : iadd, isub, imul, idiv, irem, ineg, iinc, iinc_w

****

Instructions des opérations logiques

- **Instructions de décalage arithmétique (6)**
  - short : sshl, sshr, sushr
  - int : ishl, ishr, iushr
- **Instructions des opérations booléennes (6)**
  - short : sand, sor, sxor
  - int : iand, ior, ixor

****

Instructions d'accès aux objets

- **Opérations sur des objets (35)**
  - classe : getstatic_<t>, putstatic_<t>
  - objet : getfield_<t>, getfield_<t>_w, putfield_<t>, putfield_<t>_w, new, checkcast, instanceof
  - objet courant : getfield_<t>_this, , putfield_<t>_this
  - note : $t \in \{a, b, s, i\}$
- **Operations spécifiques aux tableaux (3)** : newarray, anewarray, arraylength

****

Operations affectant le flot de contrôle (44)

- **Instructions de branchement**
  - **Branchement conditionnel (26)**
    - if<cond>, if<cond>_w, ifnull, ifnonnull, ifnull_w, ifnonnull_w, if_scmp<cond>, if_scmp<cond>_w, if_acmp<cond1>, if_acmp<cond1>_w
    - note : $cond1 \in \{eq, ne\}, cond1 \in \{eq, ne, lt, le, gt, ge\}$
  - **Branchement inconditionnel (2)** : goto, goto_w
- **Opérations de comparaison (1) :** icmp
- **Tableau de saut (4) :** stableswitch, itableswitch, slookupswitch, ilookupswitch
- **Operation d'exception (1) :** athrow
- **Clause finally (2) :** jsr, ret
- **Instructions des méthodes**
  - **Instructions d'appel (4) :** invokevirtual, invokespecial, invokestatic, invokeinterface
  - **Instructions de retour (4) :** areturn, sreturn, ireturn, return

****

Instruction sans effet (1) : nop

ANNEXE B

Concepts liés aux CSPs

Définition B.1. ***(Arité)*** *L'arité d'une contrainte $c \in C$ est le nombre de variables sur lesquelles elle porte. On dira que la contrainte est :*

- *unaire si son arité est égale à 1*
- *binaire si son arité est égale à 2*
- *n-aire si son arité est égale à n*

Définition B.2. ***(Instanciation ou configuration)*** *Une instanciation d'un ensemble de variables $P \subseteq X$ est un élément (ou tuple) I du produit Cartésien $\prod_{x_i \in P} D_i$. Elle est dite partielle si $P \neq X$ et complète sinon.*

Définition B.3. ***(Espace de recherche d'un CSP)*** *L'espace de recherche d'un CSP est l'ensemble des configurations possibles que l'on notera E, tel que : $E = D_{x1} \times D_{x2} \times ... \times D_{xn}$. L'espace de recherche est égal au produit Cartésien de l'ensemble des domaines des variables.*

Définition B.4. ***(Solution réalisable)*** *Une solution réalisable, appelée simplement une solution, pour un CSP P est une configuration complète $s \in E$ qui satisfait toutes les contraintes du problème. On note Sol(P) l'ensemble des solutions réalisables : $Sol(P) = \{s \in E \mid \forall c \in C, \text{s satisfait c}\}$*

Définition B.5. ***(Consistance de Nœud)*** *Un CSP (X, D, C) est consistant de nœud si pour toute variable x_i de X et pour toute valeur v de D_{xi} , l'affectation (x_i, v) satisfait toutes les contraintes unaires de C portant sur x_i.*

Définition B.6. ***(Consistance d'arc)*** *Soit un CSP (X, D, C), soit une contrainte binaire $c \in C$ portant sur les variables x_i et x_j avec leur domaine respectif D_{x_i} et D_{x_j}. Nous dirons que c est arc consistante si :*

- $\forall a \in D_{x_i}, \exists b \in D_{x_j}, (a, b) \in c$
- $\forall b \in D_{x_j}, \exists a \in D_{x_i}, (a, b) \in c$

Un CSP est arc consistant si toutes ses contraintes binaires sont consistantes.

Bibliographie

[ABF+03] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter et J.P. Seifert : Fault attacks on RSA with CRT : Concrete results and practical countermeasures. *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 81–95, 2003.

[AKSICL11] Ahmadou Al Khary Sere, Julien Iguchi-Cartigny et Jean-Louis Lanet : Evaluation of countermeasures against fault. *International Journal of Security and Its Applications*, 5(2):49–60, 2011.

[Arb02] Genevieve Arboit : A method for watermarking java programs via opaque predicates. *In The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, pages 102–110, 2002.

[AS02] R. Anderson et S. Skorobogatov : Optical Fault Induction Attacks. *In Workshop on Cryptographic Hardware and Embedded Systmes (CHES 2002)*, USA, 2002.

[BAFS05] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest et Darko Stefanović : Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[Bar99] Roman Barták : Constraint programming : In pursuit of the holy grail. *In Proceedings of the Week of Doctoral Students (WDS99)*, volume 4, pages 555–564. MatFyzPress Prague, 1999.

[Bar05] Roman Bartak : Constraint propagation and backtracking-based search. *Charles Universität, Prag*, 2005.

[Bar12] Guillaume Barbu : *On the security of Java Card platforms against hardware attacks.* Thèse de doctorat, Telecom ParisTech, 2012.

[BBBP13] Alessandro Barenghi, Guido M Bertoni, Luca Breveglieri et Gerardo Pelosi : A fault induction technique based on voltage underfeeding with application to attacks against aes and rsa. *Journal of Systems and Software*, 86(7):1864–1878, 2013.

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

[BBPP09] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello et Gerardo Pelosi : Low voltage fault attacks on the RSA cryptosystem. *In Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 23–31. IEEE, 2009.

[BCD08] Mark W Bailey, Clark L Coleman et Jack W Davidson : Defense against the dark arts. *ACM SIGCSE Bulletin*, 40(1):315–319, 2008.

[BDH11] Guillaume Barbu, Guillaume Duc et Philippe Hoogvorst : Java Card operand stack : fault attacks, combined attacks and countermeasures. *In International Conference on Smart Card Research and Advanced Applications*, pages 297–313. Springer, 2011.

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

[BECN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall et Claire Whelan : The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BGC+02] Luis Berrojo, Isabel González, Fulvio Corno, Matteo Sonza Reorda, Giovanni Squillero, Luis Entrena et Celia Lopez : New techniques for speeding-up fault-injection campaigns. *In Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 847–852. IEEE, 2002.

[BGL19] Etienne Boespflug, Romain Gourier et Jean-Louis Lanet : Predicting the effect of hardware fault injection. *In International Conference on Advanced Computer Science and Information Systems*, Octobre 2019.

[BGS94] David F Bacon, Susan L Graham et Oliver J Sharp : Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[BGV11] Josep Balasch, Benedikt Gierlichs et Ingrid Verbauwhede : An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. *In Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 105–114. IEEE, 2011.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet : Combined software and hardware attacks on the Java Card control flow. *In International Conference on Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011.

[BL15] Guillaume Bouffard et Jean-Louis Lanet : The ultimate control flow transfer in a Java based smart card. *Computers & Security*, 50:33–46, 2015.

[BLLL18] Sebanjila K Bukasa, Ronan Lashermes, Jean-Louis Lanet et Axel Legay : Let's shock our IoT's heart : ARMv7-M under (fault) attacks. *In Proceedings of the 13th International Conference on Availability, Reliability and Security*, page 33. ACM, 2018.

[BLM+11] Guillaume Bouffard, Jean-Louis Lanet, Jean-Baptiste Machemie, Jean-Yves Poichotte et Jean-Philippe Wary : Evaluation of the ability to transform sim applications into hostile applications. *In International Conference on Smart Card Research and Advanced Applications*, pages 1–17. Springer, 2011.

[BM08] Jean-Marie Borello et Ludovic Mé : Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[Böc12] Adrian Böckenkamp : Optimization techniques for fault-injection experiments. Support de cours, 2012. Université de Dortmund.

[Bor11] Jean-Marie Borello : *Etude du métamorphisme viral : modélisation, conception et détection.* Thèse de doctorat, Université Rennes 1, 2011.

[BOS03] Johannes Blömer, Martin Otto et Jean-Pierre Seifert : A new CRT-RSA algorithm secure against bellcore attacks. *In Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320. ACM, 2003.

[Bou14] Guillaume Bouffard : *A Generic Approach for Protecting Java Card Smart Card Against Software Attacks.* Thèse de doctorat, Limoges, 2014.

[BPS99] Sally C Brailsford, Chris N Potts et Barbara M Smith : Constraint satisfaction problems : Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[BR96] Christian Bessiere et Jean-Charles Régin : Mac and combined heuristics : Two reasons to forsake fc (and cbj) on hard problems. *In International Conference on Principles and Practice of Constraint Programming*, pages 61–75. Springer, 1996.

[BRIM98] Alfredo Benso, Maurizio Rebaudengo, Leonardo Impagliazzo et Pietro Marmo : Fault-list collapsing for fault-injection experiments. *In Annual Reliability and Maintainability Symposium. 1998 Proceedings. International Symposium on Product Quality and Integrity*, pages 383–388. IEEE, 1998.

[BS05] Arini Balakrishnan et Chloe Schulze : Code obfuscation literature survey, 2005.

[BSR10] Roman Barták, Miguel A Salido et Francesca Rossi : New trends in constraint satisfaction, planning, and scheduling : a survey. *The Knowledge Engineering Review*, 25(3):249–279, 2010.

[BTG10] Guillaume Barbu, Hugues Thiebeauld et Vincent Guerin : Attacks on java card 3.0 combining fault and logical attacks. *In International Conference on Smart Card Research and Advanced Applications*, pages 148–163. Springer, 2010.

[BVFK05] Raul Barbosa, Jonny Vinter, Peter Folkesson et Johan Karlsson : Assembly-level pre-injection analysis for improving fault injection efficiency. *In European Dependable Computing Conference*, pages 246–262. Springer, 2005.

[Cap12] Jan Cappaert : *Code obfuscation techniques for software protection.* Thèse de doctorat, University of Katholieke Leuven, 2012.

[CG10] Florence Charreteur et Arnaud Gotlieb : Constraint-based test input generation for Java bytecode. *In IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 131–140. IEEE, 2010.

[Chr13] Maria Christofi : *Preuves de sécurité outillées d'implémentations cryptographiques.* Thèse de doctorat, Versailles St Quentin en Yvelines, 2013.

[CKN14] Janusz Chorko, Tomasz Kobylarz et Piotr Nazimek : Testing fault susceptibility of java cards. *Przeglad Elektrotechniczny*, 90(2), 2014.

[CMR+01] Pierluigi Civera, Luca Macchiarulo, Maurizio Rebaudengo, Matteo Sonza Reorda et A Violante : Exploiting FPGA for accelerating fault injection experiments. *In Proceedings. Seventh InternationaL On-Line Testing Workshop*, pages 9–13. IEEE, 2001.

[CMW14] Christophe Clavier, Damien Marion et Antoine Wurcker : Simple power analysis on aes key expansion revisited. *In International Workshop on Cryptographic Hardware and Embedded Systems*, pages 279–297. Springer, 2014.

[Cor16] Marie-Angela Cornelie : *Implantations et protections de mécanismes cryptographiques logiciels et matériels.* Thèse de doctorat, Grenoble Alpes, 2016.

[CT05] Hamid Choukri et Michael Tunstall : Round reduction using faults. *FDTC*, 5:13–24, 2005.

[CTL97] Christian Collberg, Clark Thomborson et Douglas Low : A taxonomy of obfuscating transformations. Rapport technique, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CTL98] Christian Colberg, Clark Thomborson et Douglas Low : Manufacturing Cheap Resilient and Stealthy Opaque Constructs. *roc. ymp. Principles of Programming Languages (POPL'98)*, 1998.

[Dav17] Robin David : *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes*. Thèse de doctorat, Université de Lorraine, 2017.

[DBLW02] B. Den Boer, K. Lemke et G. Wicke : A DPA Attack against the Modular Reduction within a CRT Implementation of RSA. *In Cryptographic Hardware and Embedded Systems (CHES'2002)*, pages 228–243. Springer, 2002.

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre et Assia Tria : Electromagnetic glitch on the aes round counter. *In International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 17–31. Springer, 2013.

[DMN+12] Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, Assia Tria et Thierry Vaschalde : Fault round modification analysis of the advanced encryption standard. *In 2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 140–145. IEEE, 2012.

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

[Dra04] Stephen Drape : *Obfuscation of abstract data types*. Thèse de doctorat, University of Oxford, 2004.

[Dra09] Stephen Drape : Intellectual property protection using obfuscation. *Proceedings of SAS 2009*, 4779:133–144, 2009.

[Dub16] Jean Dubreuil : Java card security, software and combined attacks. *In SSTIC*, 2016.

[Dur16] Louis Dureuil : *Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute*. Thèse de doctorat, Université Grenoble Alpes, 2016.

[Eur18] EuroSmart : EUROSMART CONFIRMS ITS FORECASTS ON TRENDS AND GROWTH DRIVERS IN ALL MAIN MARKET SECTORS, Novembre 2018. `http://www.eurosmart.com/news-publications/press-release/319-eurosmart-confirms-its-forecasts-on-trends-and-growth-drivers-in-all-main-market-html`.

[Eyr17] Ninon Eyrolles : *Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools*. Thèse de doctorat, University of Paris-Saclay, 2017.

[Fau13] Emilie Faugeron : Manipulating the frame information with an underflow attack. *In International Conference on Smart Card Research and Advanced Applications*, pages 140–151. Springer, 2013.

[FBL+15] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti et Muttukrishnan Rajarajan : Android security : a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.

[FFL+16] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti et Manoj Gaur : Android code protection via obfuscation techniques : Past, present and future directions. *arXiv preprint arXiv :1611.10231*, 2016.

[FL16] Mozhdeh Farhadi et Jean-Louis Lanet : Paper tigers : an endless fight. *In International Conference for Information Technology and Communications*, pages 40–62. Springer, 2016.

[FL17] Mozhdeh Farhadi et Jean-Louis Lanet : Chronicle of a java card death. *Journal of Computer Virology and Hacking Techniques*, 13(2):109–123, 2017.

[GA03] Sudhakar Govindavajhala et Andrew W Appel : Using memory errors to attack a virtual machine. *In Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 154–165. IEEE, 2003.

[GBS14] Ekta Gandotra, Divya Bansal et Sanjeev Sofat : Malware analysis and classification : A survey. *Journal of Information Security*, 5(02):56, 2014.

[Gir07] Christophe Giraud : *Attaques de cryptosystemes embarqués et contre-mesures associées.* Thèse de doctorat, Versailles-St Quentin en Yvelines, 2007.

[GKS+12] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock et Josef Haid : Efficient fault emulation using automatic pre-injection memory access analysis. *In 2012 IEEE International SOC Conference*, pages 277–282. IEEE, 2012.

[Glo18] GlobalPlateform : Introduction to secure elements. Rapport technique, Global Plateform, Mai 2018. `https://globalplatform.org/resource-publication/introduction-to-secure-elements/`.

[GSD+08] Sylvain Guilley, Laurent Sauvage, Jean-Luc Danger, Nidhal Selmane et Renaud Pacalet : Silicon-level solutions to counteract passive and active attacks. *In FDTC*, pages 3–17. IEEE-CS, 2008.

[GT04] Christophe Giraud et Hugues Thiebeauld : A survey on fault attacks. *Smart Card Research and Advanced Applications VI*, pages 159–176, 2004.

[GWJL18] Thomas Given-Wilson, Nisrine Jafri et Axel Legay : The state of fault injection vulnerability detection. *In International Conference on Verification and Evaluation of Computer and Communication Systems*, pages 3–21. Springer, 2018.

[Hab92] Donald H Habing : The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *IEEE Transactions on Nuclear Science*, 39:1647–1653, 1992.

[Ham12] Samiya Hamadouche : Étude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité. Mémoire de Magister, Université M'hamed Bougara de Boumerdès, 2012.

[Ham14] Samiya Hamadouche : Fault enabled viruses against smart cards. poster présenté au 4ème Workshop Doctorants Xlim, Septembre 2014.

[HANR12] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi et Pradeep Ramachandran : Relyzer : Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *In ACM SIGPLAN Notices*, volume 47, pages 123–134. ACM, 2012.

[HBL+12] Samiya Hamadouche, Guillaume Bouffard, Jean-Louis Lanet, Bruno Dorsemaine, Bastien Nouhant, Alexandre Magloire et Arnaud Reygnaud : Subverting byte code linker service to characterize java card api. *In Seventh conference on network and information systems security (SAR-SSI)*, pages 75–81, 2012.

[HL13] Samiya Hamadouche et Jean-Louis Lanet : Virus in a smart card : Myth or reality ? *Journal of Information Security and Applications*, 18(2-3):130–137, 2013. DOI :10.1016/j.jisa.2013.08.005.

[HLM14] Samiya Hamadouche, Jean-Louis Lanet et Mohamed Mezghiche : Fault enabled viruses against smart cards. *In 1st symposium on digital trust in auvergne (SDTA)*, Décembre 2014. http ://confiance-numerique.clermont-universite.fr/SDTA-2014/index.php ?page=program.

[HLM19] Samiya Hamadouche, Jean-Louis Lanet et Mohamed Mezghiche : Hiding a fault enabled virus through code construction. *Journal of Computer Virology and Hacking Techniques*, 2019. DOI :10.1007/s11416-019-00340-z.

[HM09] Jip Hogenboom et Wojtek Mostowski : Full memory attack on a java card. 2009.

[HMGL14] Samiya Hamadouche, Mohamed Mezghiche, Arnaud Gotlieb et Jean-Louis Lanet : Vers une approche de construction de virus pour cartes à puce basée sur la résolution de contraintes. *Actes de la 13 ème édition d'AFADL, Atelier Francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels*, Juin 2014.

[HRL+18] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi et Ville Leppänen : Diversification and obfuscation techniques for software security : A systematic literature review. *Information and Software Technology*, 2018.

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

[Hyp03] Konstantin Hyppönen : Use of cryptographic codes for bytecode verification in smartcard environment. *Mémoire de master, Université de Kuopio*, 2003.

[ICL09] Julien Iguchi-Cartigny et Jean-Louis Lanet : Developing a trojan applets in a smart card. *Journal in Computer Virology*, 6(4):343–351, septembre 2009.

[JJK99] B. Jun, J. Jaffe et P. Kocher : Differential Power Analysis. *In Cryptology Conference on Advances in Cryptology*, pages 388–397. Springer, 1999.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli et Julie Michielin : Obfuscator-LLVM–software protection for the masses. *In IEEE/ACM 1st International Workshop on Software Protection (SPRO)*, pages 3–9. IEEE, 2015.

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

[KHEB14] Thomas Korak, Michael Hutter, Baris Ege et Lejla Batina : Clock glitch attacks in the presence of heating. *In 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 104–114. IEEE, 2014.

[KJJ+98] Paul Kocher, Joshua Jaffe, Benjamin Jun *et al.* : Introduction to differential power analysis and related attacks, 1998.

[KJJ99] Paul Kocher, Joshua Jaffe et Benjamin Jun : Differential power analysis. *In Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[KKP03] Gaurav S Kc, Angelos D Keromytis et Vassilis Prevelakis : Countering code-injection attacks with instruction-set randomization. *In Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[KMW17] Martin S Kelly, Keith Mayes et John F Walker : Characterising a CPU fault attack model via run-time data analysis. *In IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 79–84. IEEE, 2017.

[Koc96] Paul C Kocher : Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *In Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[KSV13] Duško Karaklajić, Jörn-Marc Schmidt et Ingrid Verbauwhede : Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[KT12] Xavier Kauffmann-Tourkestansky : *Analyses sécuritaires de code de carte à puce sous attaques physiques simulées.* Thèse de doctorat, Université D'Orléans, 2012.

[Kum92] Vipin Kumar : Algorithms for constraint-satisfaction problems : A survey. *AI magazine*, 13(1):32, 1992.

[Lan06] J.L. Lanet : Support de cours : Carte à puce, 2006.

[Lan12] Julien Lancia : Java Card combined attacks with localization-agnostic fault injection. *In International Conference on Smart Card Research and Advanced Applications*, pages 31–45. Springer, 2012.

[LB15] Julien Lancia et Guillaume Bouffard : Java card virtual machine compromising from a bytecode verified applet. *In International Conference on Smart Card Research and Advanced Applications*, pages 75–88. Springer, 2015.

[LB16] Julien Lancia et Guillaume Bouffard : Fuzzing and overflows in java card smart cards. *In SSTIC Conference, Rennes, France*, 2016.

[LBH+14] Michael Lackner, Reinhard Berlach, Michael Hraschan, Reinhold Weiss et Christian Steger : A fault attack emulation environment to evaluate java card virtual-machine security. *In 2014 17th Euromicro Conference on Digital System Design*, pages 480–487. IEEE, 2014.

[Lev00] Régis Leveugle : Fault injection in VHDL descriptions and emulation. *In Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 414–19. IEEE Comput. Soc, Los Alamitos, CA, USA, 2000.

[Low98] Douglas Low : Java control flow obfuscation. Mémoire de D.E.A., University of Auckland, 1998.

[LR15] Benoit Laugier et Tiana Razafindralambo : Misuse of frame creation to exploit stack underflow attacks on java card. *In International Conference on Smart Card Research and Advanced Applications*, pages 89–104. Springer, 2015.

[LS11] Da Lin et Mark Stamp : Hunting for undetectable metamorphic viruses. *Journal in computer virology*, 7(3):201–214, 2011.

[MADB+06] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert et Bart Preneel : On the effectiveness of source code transformations for binary obfuscation. *In Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, pages 527–533. CSREA Press, 2006.

[Man02] S. Mangard : A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. *In Information Security and Cryptology (ICISC'2002)*, pages 343–358. Springer, 2002.

[MDH+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson et Emmanuelle Encrenaz : Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller. *In 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.

[MDS99] T. Messerges, E. Dabbish et R. Sloan : Power Analysis Attacks of Modular Exponentiation in Smartcard. *In Cryptographic Hardware and Embedded Systems (CHES'99)*, pages 144–157. Springer, 1999.

[MLM17] Abdelhak Mesbah, Jean-Louis Lanet et Mohamed Mezghiche : Reverse engineering Java Card and vulnerability exploitation : a shortcut to ROM. *International Journal of Information Security*, pages 1–16, 2017.

[MML17] Abdelhak Mesbah, Mohamed Mezghiche et Jean-Louis Lanet : Persistent fault injection attack from white-box to black-box. *In 5th International Conference on Electrical Engineering Boumerdes (ICEE-B)*, pages 1–6. IEEE, 2017.

[MMLC11] Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet et Julien Cartigny : Smartcm a smart card fault injection simulator. *In 2011 IEEE International Workshop on Information Forensics and Security*, pages 1–6. IEEE, 2011.

[MOG01] C. Mourtel, F. Olivier et K. Gandol : ElectroMagnetic Analysis : Concrete Results. *In CHES'2001*, 2001.

[MP07] Wojtek Mostowski et Erik Poll : Testing the java card applet firewall. 2007.

[MP08] Wojciech Mostowski et Erik Poll : Malicious code on java card smartcards : Attacks and countermeasures. *In International Conference on Smart Card Research and Advanced Applications*, pages 1–16. Springer, 2008.

[MS01] Ian Miguel et Qiang Shen : Solution techniques for constraint satisfaction problems : Foundations. *Artificial Intelligence Review*, 15(4):243–267, 2001.

[NHH+17] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji et Takafumi Aoki : Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 7(1):35–46, 2017.

[Nou11] Agnès Cristèle Noubissi : *Mise à jour dynamique pour cartes à puce Java.* Thèse de doctorat, Université de Limoges, 2011.

[NSIcL09] Agnès Cristèle Noubissi, Ahmadou Al-khary Séré, Julien Iguchi-cartigny et Jean-louis Lanet : Cartes à puce : Attaques et contremesures. *In MajecSTIC 2009*, 2009.

[Ora15a] Oracle : *Java Card 3 Platform : Runtime Environment Specification, Classic Edition, Version 3.0.5.* Oracle, Mai 2015.

[Ora15b] Oracle : *Java Card 3 Platform : Virtual Machine Specification, Classic Edition, Version 3.0.5.* Oracle, Mai 2015.

[Ora15c] Oracle : *Java Card 3 Platform : Application Programming Interface, Classic Edition, Version 3.0.5.* Oracle, May 2015.

[Ort03] C Enrique Ortiz : An introduction to java card technology-part 1. *Sun Developer Network*, 29, 2003.

[Osw02] E. Oswald : Enhancing Simple Power-Analysis Attacks on Elliptic Curve Cryptosystems. *In Cryptographic Hardware and Embedded Systems (CHES'2002)*, pages 82–97. Springer, 2002.

[Ott05] Martin Otto : *Fault attacks and countermeasures.* Thèse de doctorat, Citeseer, 2005.

[PBR17] Roberta Piscitelli, Shivam Bhasin et Francesco Regazzoni : Fault attacks, injection techniques and tools for simulation. *In Hardware Security and Trust*, pages 27–47. Springer, 2017.

[PJ97] Justin Pearson et Peter G Jeavons : A survey of tractable constraint satisfaction problems. Rapport technique, Technical Report CSD-TR-97-15, Royal Holloway, University of London, 1997.

[PMPD14] Marie-Laure Potet, Laurent Mounier, Maxime Puys et Louis Dureuil : Lazart : A symbolic approach for evaluation the robustness of secured codes against control flow injections. *In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014.

[QS02] J.-J. Quisquater et D. Samyde : Eddy Current for Magnetic Analysis with Active Sensor. *In E-Smart 2002*, 2002.

[Raz16] Tiana Razafindralambo : *On the Security of micro-controllers : From smart cards to mobile devices.* Thèse de doctorat, Limoges, 2016.

[RM11] Babak Bashari Rad et Maslin Masrom : Metamorphic virus variants classification using opcode frequency histogram. *arXiv preprint arXiv :1104.3228*, 2011.

[RMI12] Babak Bashari Rad, Maslin Masrom et Suhaimi Ibrahim : Camouflage in malware : from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012.

[RNR+15] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer et Laurent Sauvage : High precision fault injections on the instruction cache of armv7-m architectures. *In 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.

[RNS+04] Klaus Rothbart, Ulrich Neffe, Ch Steger, Reinhold Weiss, Edgar Rieger et Andreas Mühlberger : High level fault injection for attack simulation in smart cards. *In 13th Asian Test Symposium*, pages 118–121. IEEE, 2004.

[SA02] Sergei P Skorobogatov et Ross J Anderson : Optical fault induction attacks. *In International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.

[SBS14] Horst Schirmeier, Christoph Borchert et Olaf Spinczyk : Rapid fault-space exploration by evolutionary pruning. *In International Conference on Computer Safety, Reliability, and Security*, pages 17–32. Springer, 2014.

[SBS15] Horst Schirmeier, Christoph Borchert et Olaf Spinczyk : Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. *In 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 319–330. IEEE, 2015.

[Sch02] Werner Schindler : Optimized timing attacks against public key cryptosystems, 2002.

[Sér10] Ahmadou Al Khary Séré : *Tissage de contremesures pour machines virtuelles embarquées.* Thèse de doctorat, Limoges, 2010.

[SGBS15] Sanjam Singla, Ekta Gandotra, Divya Bansal et Sanjeev Sofat : Detecting and classifying morphed malwares : A survey. *International Journal of Computer Applications*, 122(10), 2015.

[SGD08] Nidhal Selmane, Sylvain Guilley et Jean-Luc Danger : Practical setup time violation attacks on aes. *In Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 91–96. IEEE, 2008.

[SH07a] J.M. Schmidt et M. Hutter : Optical and EM Fault-Attacks on CRT-based RSA : Concrete results. *In Proceedings of the Austrochip*, pages 61–67. Citeseer, 2007.

[SH07b] Jörn-Marc Schmidt et Michael Hutter : *Optical and EM fault-attacks on CRT-based RSA : Concrete results.* na, 2007.

[SJPB95] D Todd Smith, Barry W Johnson, Joseph A Profeta et Daniele G Bozzolo : A method to determine equivalent fault classes for permanent and transient faults. *In Annual Reliability and Maintainability Symposium 1995 Proceedings*, pages 418–424. IEEE, 1995.

[Sko10] Sergei Skorobogatov : Optical fault masking attacks. *In Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 23–29. IEEE, 2010.

[SQ01] D. Samyde et J.J. Quisquater : ElectroMagnetic Analysis (EMA) : Measures and Countermeasures for Smart Cards. *In E-smart*, 2001.

[SS14] Ashu Sharma et Sanjay Kumar Sahay : Evolution and detection of polymorphic and metamorphic malwares : A survey. *arXiv preprint arXiv :1406.7061*, 2014.

[SS18] Jagsir Singh et Jaswinder Singh : Challenge of Malware Analysis : Malware obfuscation Techniques. *International Journal of Information Security Science*, 7(3):100–110, 2018.

[Ta10] A. Tria et al. : When Clocks Fail : On Critical Paths and Clock Faults. *In Smart Card Research and Advanced Application*, pages 182–193, 2010.

[Tav07] C. Tavernier : *Les Cartes à Puce : Théorie et mise en œuvre*. Dunod, Paris, 2e édition, 2007.

[Tsa95] Edward Tsang : *Foundations of constraint satisfaction*. Academic Press Limited, 1995.

[TSW16] Niek Timmers, Albert Spruyt et Marc Witteman : Controlling pc on arm using fault injection. *In 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

[VF10] Eric Vetillard et Anthony Ferrari : Combined attacks and countermeasures. *In International Conference on Smart Card Research and Advanced Applications*, pages 133–147. Springer, 2010.

[VTM+18] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset et S. Ermeneux : Laser induced fault injection on smartphone bypassing the secure boot. *IEEE Transactions on Computers*, 2018.

[Wag04] David Wagner : Cryptanalysis of a provably secure CRT-RSA algorithm. *In Proceedings of the 11th ACM conference on Computer and communications security*, pages 92–97. ACM, 2004.

[Win15] Stefan Winter : *On the Utility of Higher Order Fault Models for Fault Injections*. Thèse de doctorat, Technische Universität, 2015.

[Wit02] M. Witteman : Advances in smartcard security. *Information Security Bulletin*, (July):11–22, 2002.

[Wit03] M. Witteman : Java card security. *Information Security Bulletin*, 8(October):291–298, 2003.

[XZKL17] Hui Xu, Yangfan Zhou, Yu Kang et Michael R Lyu : On Secure and Usable Program Obfuscation : A Survey. *arXiv preprint arXiv :1710.01139*, 2017.

[YLMT17] Chahrazed Yahiaoui, Jean-Louis Lanet, Mohamed Mezghiche et Karim Tamine : Machine learning techniques to predict sensitive patterns to fault attack in the java card application. *JOURNAL OF EXPERIMENTAL AND THEORETICAL ARTIFICIAL INTELLIGENCE*, 2017.

[YSW18] Bilgiday Yuce, Patrick Schaumont et Marc Witteman : Fault Attacks on Secure Embedded Software : Threats, Design, and Evaluation. *Journal of Hardware and Systems Security*, pages 1–20, 2018.

[YY10] Ilsun You et Kangbin Yim : Malware obfuscation techniques : A brief survey. *In International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 297–300. IEEE, 2010.

[Zhi00] C. Zhiqun : *Java Card technologie for smart cards : Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[ZL79] James F Ziegler et William A Lanford : Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

Résumé

Les éléments sécurisés ont gagné une grande place dans notre vie quotidienne. Ils existent sous plusieurs formes. La carte à puce est l'élément le plus représentatif de la famille des éléments sécurisés. Elle est considérée comme étant un support d'exécution d'applications et de stockage d'informations très sécurisé. Vu la nature des informations qu'elles détiennent, les cartes à puce sont devenues la cible des personnes malintentionnées qui veulent s'approprier des informations sensibles qui y sont stockées voir même prendre le contrôle du système. La sécurité d'une carte à puce peut être contournée par des attaques matérielles, logicielles ou combinées. C'est dans cette dernière catégorie que s'inscrit notre travail.

Notre objectif dans cette thèse est de développer un nouveau vecteur d'attaque. En effet, c'est en maîtrisant les détails permettant de contourner la sécurité de la carte que nous pourrions par la suite trouver les contre-mesures permettant de s'en prémunir : « La meilleure défense c'est l'attaque ». La plateforme Java Card étant la plus utilisée, elle est retenue comme notre plateforme cible. Le but est de trouver une méthodologie de construction de codes malveillants activables par attaque en faute. L'idée est de cacher ce code malveillant dans un autre code sain (par construction) afin qu'il puisse être chargé dans la carte sans qu'il ne soit détecté par les mécanismes de sécurité embarqués ou une analyse du code. Une fois sur la carte, le comportement hostile est activé moyennant une injection de faute.

Pour aboutir à notre objectif, nous avons proposé deux approches complémentaires répondant chacune à un problème particulier. La première est une approche de construction de séquence de code, reliant deux états mémoire donnés, par parcours d'arbre. Elle repose sur des fondements théoriques liés au domaine des CSPs (Constraint Satisfaction Problem). La seconde approche traite le mécanisme de désynchronisation de code qui permet la dissimulation d'un code donné en opérant des transformations dessus. La mise en œuvre des deux approches a donné lieu à deux outils pouvant générer des solutions de façon automatique. Des exemples d'application et une étude de cas ont permis de présenter des exploitations possibles des approches proposées afin de réaliser des opérations mettant en danger la sécurité d'une carte à puce.

Mots-clés: élément sécurisé, Java Card, désynchronisation de code, injection de faute, satisfaction de contraintes, construction de code.

Abstract

Secure elements take place in an important part of our day to day life, they exist in several forms. The smart card is the most representative element in the family of secure elements. It is considered to be a very secured device designed to execute applications and store confidential data. Due to the nature of the information they hold, smart cards have become the target of malicious persons who want to appropriate sensitive stored information or even take control of the system. The security of a smart card can be bypassed by hardware, software or combined attacks. Our work falls in this last category.

Our objective in this thesis is to develop a new vector of attack. Indeed, it is by mastering the details that allow us to bypass the security of the card that we will then be able to find the adequate countermeasures: “The best defence is the attack”. Java Card, being the most used platform, is retained as our target platform. Our goal is to find a methodology for building malicious code that can be activated by a fault attack. The idea is to hide this malicious code into another innocent code (by construction) so that it can be loaded into the card without being detected by embedded security mechanisms or code analysis. Once on the card, the hostile behaviour is activated through a fault injection.

To achieve our objective, we proposed two complementary approaches, each responding to a specific problem. The first one is an approach for code sequence construction, linking two given memory states, by tree traversal. It is based on theoretical foundations related to the field of CSP (Constraint Satisfaction Problem). The second approach deals with the code desynchronization mechanism that allows the hiding of a given code by performing transformations on it. The implementation of the two approaches gives rise to two tools that can generate solutions automatically. Application examples and a case study presented possible ways in which the proposed approaches could be used to carry out operations that would compromise the security of a smart card.

Keywords : secure element, Java Card, code desynchronization, fault injection, constraints satisfaction, code construction.