Carte à puce Java Card: Protection du code contre les attaques en faute

Ahmadou Al Khary Séré, Jean-Louis Lanet, Julien Iguchi-Cartigny

Université de Limoges, Laboratoire XLIM, Équipe SSD, 83 rue d'Isle, 87000 Limoges - France.

Contact : {ahmadou-al-khary.sere, jean-louis.lanet,julien.cartigny,}@xlim.fr

---

Résumé: La sécurité des cartes à puce est souvent mise à rude épreuve par les attaquants qui utilisent tous les procédés matériels ou logiciels possibles. Le but étant de pouvoir accéder aux informations et aux secrets contenus dans la carte (code PIN, clé(s) secrète(s) cryptographique(s), etc.) ou tout simplement de nuire aux applications embarquées afin de tester le niveau sécuritaire de la carte. En effet même pour les cartes modernes, ces attaques représentent une vraie menace. Dans cet article, nous présentons quelques contremesures pour des attaques en fautes. Nous proposons notamment des protections permettant d'instrumenter le programme afin de pouvoir détecter les modifications pouvant avoir lieu lors de son exécution.

Mots clefs: ***Java, Java Card, Attaque en faute, Byte code, Graphe de flux de contrôle***

1 Introduction

Boneh, DeMillo et Lipton ont proposé en 1996 un nouveau modèle d'attaque sur carte à puce qu'ils ont appelé cryptanalyse en présence de fautes matérielles [1][3]. Une attaque en faute consiste à perturber l'environnement physique de la carte lors de son fonctionnement. Ces perturbations peuvent être produites de différentes manières : variations de la tension d'alimentation ou de la fréquence d'horloge de la carte, émissions électromagnétiques, utilisation d'un courant de Foucault ou d'une émission laser proche de la puce.

Ces perturbations introduisent des erreurs sur la puce qui perturbent les données utiles (mémoires, pointeur de code, registres, etc.) lors de l'exécution du code de la carte. En particulier, une telle attaque peut modifier le flux de contrôle d'un programme.

Cette attaque visait initialement quelques algorithmes cryptographiques à clef publique comme le système de signature RSA et le système d'authentification de Fiat-Schamir et Shnorr. Biham et Shamir ont également montré en [2] que le DES était potentiellement vulnérable à ce type d'attaque.

Cependant, l'impact de ce type d'attaque va au-delà de l'implémentation des algorithmes cryptographiques. Il concerne l'ensemble des programmes embarqués dans la carte et peut cibler n'importe quelle partie de code sur laquelle repose la sécurité de la carte à puce. Cette classe d'attaque peut par exemple être utilisée pour éviter le contrôle par code PIN ou outrepasser les vérifications des mécanismes de protection logiciels déployés sur la puce. Elle constitue donc à l'heure actuelle une sérieuse menace pour la sécurité des cartes à puce.

Au cours de notre travail, nous nous sommes intéressés plus particulièrement aux cartes à puce de type Java Card. Cette technologie fournit une machine virtuelle permettant à plusieurs applications (*Applets*) de s'exécuter sur la carte dans un environnement sécurisé. Le langage Java Card est un sous-ensemble de Java, il fonctionne à ce titre de la même façon : une machine virtuelle à pile, un environnement d'exécution et une API de programmation.

Après la compilation d'une *applet* Java Card 3.0 [3], on obtient un fichier cap (classic edition) ou un fichier class (connected edition) . Dans ces fichiers, le code intermédiaire (*byte code*) contient toutes les informations nécessaires à l'exécution d'une application. Lors de l'exécution dans la machine virtuelle, le pointeur de code va exécuter séquentiellement chaque instruction (*opcode*) en utilisant une pile des opérandes. Un *opcode* va chercher ses opérandes sur cette pile le cas échéant et y déposera son résultat.

Dans le cadre de ce travail, nous nous sommes intéressés à la sécurité du fichier class avec en tête la problématique principale de l'intégrité du code exécuté sur la carte, c'est-à-dire être sûr que le code exécuté sur la carte est bien celui qui a été chargé.

Le reste de cet article va s'articuler de la façon suivante : dans la suite de cette section, nous décrirons le principe général d'une attaque en fautes. Dans la section 2, nous évoquerons les différents modèles de faute possibles sur les cartes à puce. La section 3 présente les contre-mesures matérielles et logicielles déjà existantes. La section 4 propose notre contribution et nous l'évaluerons contre une attaque logicielle.

1.1 Descriptions de quelques attaques en faute

1.1.1 Attaque électrique

C'est une attaque basée sur la variation de l'alimentation de la carte. Le but étant d'introduire une faute sans toucher à l'intégrité physique de la carte. En effet, le comportement du circuit électrique est défini pour des plages d'alimentation bien définies, en dehors de ces plages, le comportement de la puce est imprévisible. Dans certains cas, cela peut être suffisant pour introduire une faute [4]. Un pic énergétique peut varier selon 9 critères différents incluant sa hauteur, sa forme, sa durée, etc. On peut réaliser un pic en influant uniquement sur la source d'énergie (le lecteur de cartes par exemple), il n'y a donc pas besoin d'avoir

un accès direct à la puce. Le matériel requis pour mener à bien cette attaque dépend du type de pic que l'on souhaite obtenir, mais n'est pas nécessairement couteux.

1.1.2 Attaque utilisant la fréquence d'horloge

Elle est basée sur la perturbation de la vitesse d'horloge qui s'applique facilement aux cartes ne disposant pas de PLL (Phase-Locked Loop). Un exemple d'utilisation de cette attaque est le cas où une mise à jour est faite à une vitesse deux fois plus élevée, alors il se peut que certaines instructions soient affectées tandis que d'autres ne le seront pas, en conséquence on va provoquer l'exécution prématurée de certaines instructions qui vont utiliser des données anciennes à la place des données attendues [5]. L'introduction de ces perturbations peut influencer les branchements conditionnels (en les effectuant ou non selon le comportement désiré), un registre peut être décalé deux fois ou pas du tout.

1.1.3 Attaque optique

En concentrant une lumière avec une longueur d'onde spécifique, il est possible d'inverser le contenu d'une cellule mémoire. Cela permet de modifier la mémoire en utilisant l'effet photoélectrique. Ces attaques requièrent d'être capables d'atteindre la puce, elle nécessite donc un retrait des protections physiques de la carte. Il a été montré en [6] que le matériel nécessaire pour réaliser ces attaques est relativement peu couteux et qu'elles peuvent également être très précises en affectant un seul bit.

1.1.4 Attaque en utilisant les perturbations électromagnétiques

En créant un fort champ électromagnétique à proximité de la mémoire, les ions représentant son état vont bouger et par conséquent vont entrainer sa perturbation. Il a été prouvé en [7] que le courant de Foucault pouvait être créé en utilisant une bobine active avec suffisamment de puissance. Elle nécessite un matériel peu couteux et elle permet d'avoir un contrôle très précis des bits touchés.

2 Modèle d'attaque retenu

Dans un premier temps, il est nécessaire d'évaluer le pouvoir d'un attaquant. Il existe un grand nombre d'attaques en faute dans la littérature. Elles se différencient par leur localisation (nombres de bits affectés par exemple), leur précision (le contrôle sur le nouvel état qu'engendre la faute), et la fenêtre temporelle d'opportunité pour l'attaque.

De la littérature (notamment [16] et [17]), on peut principalement retenir quatre modèles de faute :

- **Modèle de faute nº 1 :** L'attaquant peut causer une faute sur un seul bit avec un contrôle total sur la localisation de ce bit et l'instant opportun pour appliquer la faute.
- **Modèle d'attaque nº 2** : L'attaquant peut causer une faute sur un octet unique avec un contrôle total la localisation et l'instant opportun pour appliquer la faute.
- **Modèle d'attaque nº 3** : l'attaquant peut causer une faute sur un octet unique avec un contrôle partiel sur la localisation et l'instant opportun pour appliquer la faute. De plus, il est incapable de prédire la nouvelle valeur erronée introduite.
- **Modèle d'attaque nº 4** : l'attaquant peut causer une faute sur une variable unique avec un contrôle partiel sur la localisation et l'instant opportun pour appliquer la faute. De plus, la variable cible va être remplacée par une valeur aléatoire inconnue par l'attaquant.

Ces modèles sont simplistes et ne sont montrés ici que pour illustrer notre choix dans les pouvoirs de l'attaquant. Il existe en effet d'autres paramètres comme le taux de réussite d'une attaque (dont on peut déduire un nombre moyen d'attaques successives avant qu'une attaque réussisse), du coût technique d'une attaque (principalement le coût matériel), et des possibilités de faire deux attaques successives

(éventuellement avec une localisation différente).

Dans le cadre de ces attaques, la carte à puce peut réagir de différentes façons. Les cartes non protégées ne vont pas déceler l'attaque. Les cartes les plus sophistiquées peuvent avoir des protections permettant d'obtenir des valeurs aléatoires en guise de résultat ou permettant de détecter une erreur. Les cartes modernes hauts de gamme vont quant à elles être équipée de protections matérielles supplémentaires comme en [8], qui peuvent contrer efficacement ce type d'attaque.

Les différents modèles d'attaque sont cités par ordre décroissant en terme de puissance d'attaque entrainant par la même que si l'on se protège contre un modèle de faute, on se protège aussi contre les fautes ayant un rang plus élevé (c'est-à-dire que si l'on est protégé contre le modèle nº 1 on est également protégé contre les modèles 2, 3 et 4). En général, le modèle d'attaque nº 1 est difficile à reproduire, car la plupart des cartes à puce fiables contiennent différentes protections rendant difficile la localisation d'un seul bit de manière précise. Raison pour laquelle, nous avons choisi de retenir le modèle d'attaque nº 2.

Les attaques réalisées avec le modèle de faute choisi permettent d'avoir divers effets sur la carte. Ces effets peuvent être résumés comme suit :

- La randomisation des données : l'adversaire peut changer les données à des valeurs aléatoires qu'il ne contrôle pas.
- La réinitialisation des données : l'adversaire peut forcer le passage des données à la valeur neutre c'est-à-dire la réinitialisation d'un octet précis aux valeurs 0x00 ou 0xFF.
- La modification d'*opcodes* : l'adversaire peut modifier les instructions exécutées par le processeur de la puce. Cela aura les mêmes effets que les deux cas précédents. Les effets supplémentaires peuvent inclure une suppression de fonction ou l'arrêt de boucle. Les deux précédents effets sont dépendants de l'algorithme tandis que la modification d'*opcodes* est dépendante de l'implémentation.

3 État de l'art des mécanismes de protection

Il existe plusieurs méthodes de protection contre les attaques en faute. Elles peuvent être matérielles ou logicielles. Certaines ont pour rôle de détecter des conditions anormales de l'environnement (détecteur de lumière ou de température) ou sur les entrées de la puce (variation hors-norme de la fréquence de l'horloge ou du voltage d'alimentation). D'autres utilisent des dispositifs pour empêcher un attaquant d'avoir une information exploitable (chiffrement de certains bus et/ou d'une partie de la mémoire, ajouts de couches métalliques). Enfin, il existe aussi des méthodes consistant à vérifier le bon comportement de la puce. On peut évoquer par exemple la redondance d'exécution lors d'opérations critiques.

Dans cet article, nous proposons des contre-mesures logicielles permettant d'avoir des mécanismes de défense plus flexibles, mais moins spécialisés que leurs pendants matériels. Elles peuvent remplir les mêmes fonctions que certaines contre-mesures matérielles, par exemple pour la redondance d'exécution. Mais en général, elles sont utilisées dans des tâches de défense du logiciel embarqué dans la carte.

La littérature la plus conséquente sur ce propos concerne les attaques en faute sur les opérations cryptographiques de la carte (comme le prouve [20][16][9]). À partir de l'injection de fautes, un attaquant peut réduire la sureté d'un algorithme cryptographique ou déduire des informations permettant de découvrir certains secrets enfouis. Les contre-mesures associées peuvent détecter des comportements anormaux de l'algorithme en vérifiant certains paramètres durant l'exécution des algorithmes à protéger. Elles peuvent aussi consister à modifier les algorithmes afin que les attaques en fautes ne dévoilent aucune information sensible. Mais en général, ces contre-mesures sont spécialisées, car elles ont une connaissance de la sémantique des données associées. Notre approche s'intéresse plutôt à la sémantique et à l'interprétation d'un autre type de donnée : le code. Le but recherché étant de pouvoir vérifier que le code stocké dans l'EEPROM, après édition de liens, est bien celui qui est exécuté.

3.1 AGL

Akkar, Goubin et Ly (AGL) ont proposé en 2003 [10] un procédé pour lutter contre les attaques en faute. Pour cela, AGL vérifie le graphe de flux de contrôle en certains points de l'exécution à partir de séquence d'exécution valide indiqué par le programmeur. Ce dernier écrit son application et précise les fonctions à protéger. Il utilise des directives de *preprocessing* pour marquer le début et les différents points de passage de la zone à protéger. Il indique par la suite la fin de cette zone à défendre où la vérification sera effectuée. Ainsi, durant la phase de compilation du programme, un outil va calculer l'ensemble des chemins possibles dans la zone critique et en déduire des séquences valides de points de passage. Puis, durant l'exécution, à chaque fin de zone critique, le système comparera l'historique de points de passage vis-à-vis des séquences valides.

Afin de limiter les informations à mémoriser et le surcoût en terme de traitement [10], le programmeur peut choisir la granularité de la protection en indiquant la structure de données mémorisant l'historique d'exécution de la zone critique:

- soit un compteur indiquant le nombre de points de passages appelés,
- soit un champ de bits indiquant quels sont les points de passage appelés au moins une fois,
- soit une pile mémorisant l'historique complet des points de passage.

3.2 La technique des blocs élémentaires

Un brevet [11] a pour but de protéger les applications sur une carte des manipulations non autorisées. La méthode proposée utilise un partitionnement du code de l'application en plusieurs blocs élémentaires. Un bloc élémentaire est une unité atomique de code qui a un point d'entrée, un point de sortie et un ensemble d'unité de données. Dans notre cas, les unités de données vont correspondre à l'ensemble des instructions du bloc. À chaque bloc élémentaire, une valeur de contrôle associée est calculée. Cette valeur est fonction des unités de données composant ce bloc. Cette valeur est ensuite sauvegardée et réutilisée a posteriori pour vérifier si pendant l'exécution du bloc élémentaire ou si avant son exécution celui-ci n'a pas été modifié. Pendant ou avant l'exécution du bloc élémentaire, la valeur de contrôle est à nouveau calculée et comparée avec celle sauvegardée. Si elles ne sont pas identiques, c'est qu'il y aura eu une erreur. Dans ce brevet, la technique proposée afin de calculer les valeurs de contrôle est un MD5 ou un SHA-1 utilisant tous les éléments qui constituent le bloc. Le problème vient des algorithmes proposés pour calculer les valeurs de contrôle. En effet, le nombre d'opérations pour calculer un MD5 ou un SHA-1 est très important donc le coût pour la carte en ressource processeur va être élever, sachant qu'il va falloir refaire ces opérations autant de fois qu'il y a de blocs dans la méthode. Ensuite, les auteurs ne précisent pas s'il y a un moyen de filtrer les méthodes à protéger ou non. Nous supposons donc qu'il faut le faire sur toutes les méthodes de l'*applet* augmentant par la même le nombre de fois que l'on va calculer les valeurs de contrôle.

3.3 Intégrité des flux de contrôle

En [12], il est question d'un mécanisme permettant de vérifier l'intégrité des flux de contrôle appelé CFI pour « Control Flow Integrity ». CFI se base sur le graphe de flux de contrôle pour s'assurer de l'exécution correcte du programme. Ce CFG est construit en analysant le fichier binaire. Le principe est de vérifier lorsqu'il y a un transfert du flux de contrôle (un saut conditionnel par exemple) que ce transfert a bien lieu vers une destination valide. Pour mener à bien cette vérification, les fichiers binaires sont réécrits. À chaque fois qu'il y a un saut, la destination est marquée à l'aide d'un *label* unique et la source est modifiée

de sorte à intégrer une vérification. Ainsi, avant d'effectuer le transfert, on va vérifier que la destination correspond bien à une destination autorisée en vérifiant la liste des *labels* associés.

4 Proposition de mécanismes de protection

4.1 Illustration

Pour illustrer nos propositions, nous allons utiliser un exemple concret avec une méthode d'une *applet* Java Card. Cette *applet* a pour nom JavaPurse et gère un portemonnaie électronique. La méthode choisie est la suivante :

```
private void processSelectFile(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
     // get the apdu data
     apdu.setIncomingAndReceive();
     if (buffer[ISO7816.OFFSET_P1] == (byte)2) {
        //  select file by FID
        if ( buffer[ISO7816.OFFSET_LC] != (byte)2)
          ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        short fid = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
        switch (fid) {
        case PARAMETERS_FID:
        case TRANSACTION_LOG_FID:
        case BALANCES_FID:
           transientShorts[SELECTED_FILE_IX] = fid;
           break;
        default:
           ISOException.throwIt(ISO7816.SW_FILE_NOT_FOUND);
        }
     } else
        ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
  }
```

Le code intermédiaire équivalent à cette méthode est le suivant :

Private void processSelectFile (javacard.framework.APDU);	Code : 0: aload_1 1: invokevirtual #34;

```
4:  astore_2
5:  aload_1
6:  invokevirtual   #52;
9:  pop
10: aload_2
11: iconst_2
12: baload
13: iconst_2
14: if_icmpne       83
17: aload_2
18: iconst_4
19: baload
20: iconst_2
21: if_icmpeq       30
24: sipush  26368
27: invokestatic    #40;
30: aload_2
31: iconst_5
32: invokestatic    #53;
35: istore_3
36: iload_3
37: tableswitch{ //-28414 to -28412
            -28414: 64;
            -28413: 64;
            -28412: 64;
            default: 74 }
64: aload_0
65: getfield  #14;
68: iconst_5
69: iload_3
70: sastore
71: goto    80
74: sipush  27266
77: invokestatic    #40;
80: goto    89
83: sipush  27265
86: invokestatic    #40;
89: return
```

4.2 Le point de vue du programmeur

Un programmeur est la personne la mieux placée pour connaître les sections critiques de code à protéger. Nous proposons d'utiliser un mécanisme pour tagger son code sur lequel des contre-mesures sont nécessaires. Du point de vue du programmeur, celui-ci va créer son *applet* Java Card dans laquelle, il va préciser les méthodes à protéger grâce à des annotations Java (afin d'éviter l'utilisation globale de ce mécanisme à tout le code Java Card contenu dans la carte). Les annotations java (disponible à partir de Java 5) permettent d'ajouter des métas-données à un code source Java. Nous proposons de fournir une annotation @Secure contenant un paramètre précisant la méthode de sécurité à appliquer sur la méthode. Celle-ci sera définie de la façon suivante:

```
@Retention{RetentionPolicy.CLASS}
public @interface secure {
        type security();
        public static enum type (FIELDOFBYTE, BASICBLOC, ...)
}
```

Ainsi, pour protéger la méthode methodToProtect avec le mécanisme de sécurité FIELDOFBYTE, le programmeur écrira :

```
@Secure(security=FIELDOFBYTE)
public void methodtoProtect() {
...
```

En interne, cette annotation est conservée dans la partie constant pool de chaque fichier class. Elle sera exploitée lors de la génération du fichier cap (dans le cas de JavaCard 3.0 classic edition) ou pour une manipulation du fichier class avant envoi à la carte (dans le cas de JavaCard 3.0 connected edition).

Nous présentons dans la suite deux méthodes utilisant cette annotation.

4.3 La méthode du champ de bit

L'idée de base de cette contre-mesure vient du constat que si une attaque modifie un *opcode* par un autre il est possible que le nombre d'opérandes ne soit plus cohérent avec le nouvel *opcode*. Plus précisément, on peut obtenir les situations suivantes :

- Soit une augmentation du nombre d'opérandes de l'*opcode,* par exemple lors du remplacement de ADD (zéro opérande) par ICMPEQ (deux opérandes).
- Soit une diminution du nombre d'opérandes de l'*opcode,* par exemple lors du remplacement de ALOAD (un opérande) par l'*opcode* ATHROW (zéro opérande).
- Soit un nombre d'opérandes de l'*opcode* inchangé, par exemple lors du remplacement de ILOAD (un opérande) par l'*opcode* RET (un opérande).

Nous proposons une méthode détectant les décalages provoqués par les changements du nombre d'opérandes. Lors de la compilation, un champ de bits est généré représentant le type de chaque entrée dans le *byte code* d'un programme. La valeur de contrôle 1 est utilisée pour un *opcode* et la valeur 0 est utilisée pour un opérande. Par exemple, pour les lignes 0 à 9 du code illustratif:

code			Valeur de contrôle
0:	aload_1		1
1:	invokevirtual	#34;	1 0 0
4:	astore_2		1
5:	aload_1		1
6:	invokevirtual	#52;	1 0 0
9:	pop		1

Avant le chargement dans la carte, un outil va analyser le contenu de chaque fichier class et s'il trouve une annotation @secure, il va appliquer notre contre-mesure à la méthode associée. Il ajoute alors un nouveau composant dans le fichier contenant le tableau de bits associé à la méthode et référence ce composant dans le constant pool.

Lorsque la méthode protégée est exécutée dans la carte, un mécanisme va vérifier la cohérence de tableau de bits avec les appels d'*opcode, c.-à-d.* pour chaque *byte code* traité, la machine virtuelle vérifie dans le tableau de bit sauvegardé que la valeur de contrôle est cohérente.

L'utilisation de cette méthode a l'avantage de nécessiter une surcharge faible pour la puce. En terme de puissance de calcul, le surcoût ne représente que quelques opérations simples sur le processeur pour chaque *opcode*. En quantité de mémoire, nos premiers tests sur plusieurs *applets* Java Card dans le cas pire (c'est-à-dire la protection de toutes les méthodes) ont montré une augmentation de la taille du *byte code*

d'environ 3% ce qui est un surcoût acceptable pour une carte à puce.

4.4 Contre-mesure basée sur les blocs élémentaires

Cette protection s'inspire de [12] (détaillé en 3.2). Son principe repose sur la subdivision du code en plusieurs blocs élémentaires et sur le calcul d'une valeur de contrôle. Mais dans notre cas, nous allons utiliser une autre fonction moins complexe pour calculer les valeurs de contrôle.

Dans un premier temps, notre outil calcule les différents blocs élémentaires qui composent le code d'une *applet*. Nous proposons d'utiliser la même méthode décrite dans [15]: nous allons rechercher les instructions pouvant commencer ou terminer un bloc élémentaire. Ces instructions vont s'appeler des *leaders* dans la suite de l'exposé. Ces *leaders* vont être déterminés grâce aux hypothèses suivantes :

1. Toute instruction qui débute une méthode
2. Toute instruction cible d'un branchement inconditionnel (goto, goto_w, jsr, jsr_w et ret).
3. Toute instruction cible d'un branchement conditionnel (ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne, lcmp, fcmpl, fcmpg, dcmpl, dcmpg).
4. Toute instruction cible d'un branchement conditionnel composé (lookupswitch et tableswitch).
5. Toute instruction qui suit une instruction leader de type 2, 3 ou 4
6. Toute instruction qui suit les instructions du type return (ireturn, lreturn, freturn, dreturn, areturn, and return).

Chacun des leaders donne naissance à un bloc élémentaire comprenant l'ensemble des instructions jusqu'au prochain leader ou jusqu'à la fin du *byte code* de la méthode. Nous proposons de plus certaines optimisations afin de limiter le nombre de blocs élémentaires et par la même occasion le nombre de point de vérification. Par exemple, l'intégration des invocations de méthode dans les blocs élémentaires, ou celle des instructions capables de lever des exceptions.

Si nous appliquons les six points précédents à notre méthode processSelectfile, les leaders vont être les instructions au PC 0, 14, 17, 21, 24, 37, 64, 71, 74, 80, 83, 89. Les blocs élémentaires sont donc : (0,14), (17,21), (24,37), (64,71), (74,80) et (83,89).

Pour chaque bloc élémentaire est calculée une valeur de contrôle mémorisée avec les informations de début et de fin de celui-ci. Ces informations sont enregistrées dans un composant du fichier class référencé par le constant pool sous forme d'une table. Nous proposons d'utiliser l'opérateur XOR, comme en [15], afin de calculer cette valeur de contrôle. Ainsi, chaque fois que le PC va être incrémenté, la machine virtuelle (VM) va regarder dans la table sauvegardée, s'il correspond à un début ou une fin de bloc. Si elle rencontre un début de bloc (dans la table), elle va commencer à faire l'opération de signature (XOR) jusqu'à ce qu'elle rencontre la fin de ce bloc (dans la table). Une vérification supplémentaire est faite au niveau du contenu du PC en début et en fin de bloc, ainsi dans chacun de ces cas, la machine virtuelle va vérifier que le contenu du PC correspond bien à un *leader*. Quand une de ces vérifications échoue, on arrête l'exécution du programme. Sur le premier bloc calculé précédemment : (0,14) avec une valeur de contrôle 4D, la table sauvegardée contiendra un trio {0, 14, 4D}. La machine virtuelle va rencontrer le PC 0, elle va vérifier que ce PC est bien dans notre table et que son contenu est bien une instruction pouvant débuter un bloc, puis l'opération de signature va commencer jusqu'à la rencontre du PC 14, si ce PC est bien une instruction pouvant terminer un bloc, alors le résultat est comparer avec la valeur 4D. Si ces deux valeurs sont identiques, alors on réinitialise la valeur de la signature et on continue l'exécution du programme avec le bloc suivant, sinon on l'arrête.

Les modifications à la chaîne de développement d'*applet* Java Card sont légères. Comme en 4.3, le programmeur utilise la même annotation avec un nouveau paramètre : @secure (security=BASICBLOC). De

plus, il est nécessaire de créer un composant additionnel contenant un octet pour coder le début de bloc élémentaire, un octet pour la sa fin et un octet pour sa signature. Le fichier class a alors un surcoût en terme de taille dans le pire des cas entre 1% et 5%, valeurs qui nous semblent acceptables dans le cas de la carte à puce. L'écart important dans cette estimation s'explique du fait que la taille du composant va être fonction du nombre de blocs élémentaires contenu dans le fichier class (tandis que dans la méthode 4.3, elle est proportionnelle aux nombres d'instructions de ce fichier).

À la différence de la contre-mesure 4.3, celle-ci est plus complexe et fait une vérification a posteriori de la valeur du code. Toutefois, cette complexité nous permet de nous prémunir contre des attaques (suivant le modèle de faute retenu) pouvant mener à la modification des *opcodes* avec le même nombre d'opérandes.

. Pour la phase de test, nous avons conçu et développé un interpréteur abstrait de *byte code* permettant ainsi de tester l'impact des attaques et d'évaluer le coût des modifications nécessaires pour l'ajout d'un tableau de bit dans un fichier class. D'autres travaux sont en cours afin de permettre le test de ces contre-mesures sur une Java Card proprement dite.

4.5 Évaluation de la méthode.

Nous avons évalué l'efficacité de cette méthode face à l'attaque EMAN [21]. Cette attaque n'est pas une attaque matérielle, mais une attaque logique dont l'effet est similaire à une attaque en faute puisqu'elle permet de remplacer des portions de code dans une *applet* appartenant à un autre contexte de sécurité. Elle agit donc comme un cheval de Troie possédant un code auto modifiable nous permettant de lire et écrire n'importe où dans la mémoire de la carte. Nous avons implémenté une fonction de recherche et de remplacement de motifs afin de remplacer des parties de code dans la carte. Afin de montrer la puissance de cette attaque, considérons le code suivant très souvent utilisé pour la vérification du PIN code dans une Java Card. Il est basé sur l'utilisation de l'API Java Card et de l'objet *OwnerPin* qui est une implémentation sécurisée (décrémentation du compteur de ratification avant la comparaison, etc.) de l'objet modélisant un Pin code. Dans le fragment de code suivant, lorsque l'*applet* veut réaliser un débit, elle se prémunit en vérifiant si le Pin code a déjà été validé auparavant en utilisant la méthode *isValidated ()* sur l'objet Pin code. Ce fragment de code est très classique dans son utilisation. Si l'utilisateur entre un mauvais code alors une exception est émise et la fonction est abandonnée.

```
public void debit (APDU apdu )
                    {
                     ...
                     if (!pin.isValidated())
                       {
                        ISOException.throwIt(SW_AUTH_FAILED)
                       }
                       // do safely something authorized
                    }
```

L'objectif de notre cheval de Troie est de rechercher dans le code de cette application (qui évidemment n'appartient pas au contexte de sécurité de l'attaquant et ne lui est même pas accessible en lecture) le *byte code* traitant l'exception. Par exemple, si notre attaque détecte en mémoire le code 11 69 85 8D xx xx et si le propriétaire du code est l'*applet* ciblée alors il suffit de remplacer ce code par le motif suivant : 00 00 00 00 00 00. Le *byte code* 00 étant celui qui code l'instruction NOP, le fragment de code de l'*applet* chargée devient :

```
public void debit (APDU apdu )
{
...
if (!pin.isValidated())
```

```
{ }
// do safely something authorized
}
```

L'intérêt de cette fonction de recherche et remplacement devient évident : il est possible de contourner les fonctions de sécurité de n'importe quel *applet* chargée dans la carte. Si le cheval de Troie est capable de lire et écrire en mémoire, il devient aussi possible de l'utiliser pour caractériser la représentation des objets pour la machine virtuelle embarquée. Il devient aussi possible d'obtenir le code d'implémentation des algorithmes cryptographiques selon l'espace mémoire où ils sont implémentés, lequel peut à son tour générer de nouvelles attaques.

Si le développeur de l'*applet* attaqué a inséré une annotation du type @secure (security= BASICBLOC) alors la méthode sera protégée par la technique des blocs de base. Lorsque la machine virtuelle va interpréter le code correspondant, elle va détecter à plusieurs moments la modification du code original.

La fin du bloc de base correspondant à la séquence d'instructions sspush; if_acmpne #85 sera modifiée et donc la signature sera erronée. Lorsque le pointeur de programme arrivera à l'adresse de fin du bloc il va détecter que le *byte code* correspondant doit être un test ou toute instruction terminant un bloc, or ce dernier a été remplacé par le *byte code* NOP. Donc, outre la signature erronée, le bloc a une mauvaise terminaison. Le second point concerne le début de bloc suivant. Dans la table des blocs, le PC détecte un début et doit procéder à la ré-initialisation de la variable de signature et procède à la signature des trois premiers octets. Ces derniers ayant été changés la signature de fin de bloc sera donc elle aussi erronée.

On voit ainsi que la modification de la machine virtuelle permet de détecter non seulement des attaques matérielles, mais aussi des attaques logiques ayant pour but de modifier la mémoire où résident les codes des applications. Dans cet exemple, on voit que le mécanisme utilise trois éléments successifs pour détecter la modification : deux concernent la signature et le troisième concerne l'instruction de terminaison de bloc.

5 Conclusion

La grande majorité de la littérature sur les attaques par fautes se consacre uniquement au problème de détournement des algorithmes cryptographiques. Mais ces attaques peuvent aussi servir à modifier des comportements beaucoup plus basiques dans du code, qu'il soit natif ou interprété. Dans le cas de la Java Card, ces attaques ont d'ailleurs une bien meilleure faisabilité par la connaissance possible par l'attaquant de la structure d'une partie de la mémoire (en général identique à celle d'un fichier cap). Disposer de mécanismes de protection simples pour le programmeur et peu coûteux en terme de mémoire et en temps de traitement offre la possibilité de protéger la partie code d'un programme sans remettre en question la chaîne de développement des *applets* Java Card. Néanmoins, cette approche a une limite, car elle exploite la structure connue du code contenu dans une machine virtuelle, mais ne peut raisonner sur la signification des zones de données d'une carte. Un attaquant a donc par exemple toujours la possibilité d'attaquer les algorithmes cryptographiques en modifiant seulement les données.

6 Références bibliographiques

[1]. D. Boneh, R. DeMillo, R. Lipton, *New Threat Model Breaks Crypto Codes. Bellcore Press Release*, September 25th, 1996.

[2]. E. Biham, A. Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*. In Proceedings of CRYPTO'97, Lecture Notes in Computer Science, Vol. 1294, Springer, pp. 513-528, 1997.

[3]. Sun Microsystem, *Java Card 3.0 (TM) spécifications*, 28 mars 2008.

[4] C. Aumuller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert *Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures* , Proceedings of CHES '02, LNCS 2523, Springer-Verlag.

[5] Oliver Kommerling, Markus G. Kuhn. *Design Principles for Tamper- Resistant Hardware*, USENIX workshop on Smartcard Technology 1999.

[6] Sergi P. Skorobogogatov, Ross J.Anderson *Optical Fault Induction Attacks* , Proceedings of CHES '02, LNCS 2523, Springer-Verlag.

7] J.-J. Quisquater, D. Samyde *Eddy Current for Magnetic Analysis with Active Sensor*, E-Smart 2002, NOVAMEDIA.

[8] J. Blomer and J.-P. Seifert. *Fault based cryptanalysis of the Advanced Encryption Standard (AES)*. In Seventh International Financial Cryptography Conference (FC 2003) (Gosier, Guadeloupe, FWI January 27-30), 2003.

[9] A. Shamir. *Method and apparatus for protecting public key schemes from timing and fault attacks*, 1999. US Patent No. 5,991,415, Nov. 23, 1999.

[10] M-L. Akkar, L. Goubin, Olivier Ly *Automatic integration of counter-measure against fault injection attacks*, 2003

[11] S. Prevost, K. Sachdeva, *Application Integrity Check During Virtual Machine Runtime*, 2006, US Patent No. 20060047955A1, Mar. 2, 2006.

[12] M. Abadi, M. Bidiu, U. Erlingsson, J. Ligatti, *Control Flow integrity*, Proceedings of the 12th ACM Conference on Ccomputer and communications security, 2005.

[13] T. Lindholm, F. Yellin, *The Java(TM) Virtual Machine Specification*, Second Edition, April 24, 1996

[14] J. Zhao, *Analyzing Control Flow in Java Bytecode*, Prooceedings of 16th Conference for Japan Society for Software Science and Technology, pp 313-316, 1999.

[15] N.Oh, P. P. Shirvani, E. J.McCluskey, *Control Flow Checking by software signature*, IEEE transaction on reliability, Vol. 51, N°2, March 2002.

[16] J. Blömer, M.Otto, J-P. Seifert, *A new CRT-RSA Algorithm Secure Against Bellcore Attacks*, Proceedings

of 10th ACM Computer and communications security, pp 311-320, 2003.

[17] D. Wagner, *Cryptanalysis of a provably secure CRT-RSA algorithm*, Prooceedings of the 11th ACM conference on computer and communications security, pp 92-97, 2004.

[18] P. Dusart, G. Letourneux, O. Vivolo. "*Differential Fault Analysis on A.E.S.*", Cryptology ePrint Archive: Report 2003/010. `http://www.iacr.org`.

[19] Ch. Giraud, *"DFA on AES"*, Cryptology ePrint Archive: Report 2003/008. `http://www.iacr.org`.

[20] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, D.T Ltd, I. Rehovot, *The sorcerer's apprentice guide to fault attacks,* Proceedings of the IEEE, Vol. 94, N°2, pp 370-382, 2006.

[21] J. Iguchi-Cartigny, J.-L. Lanet *Évaluation de l'injection de code malicieux dans une Java Card* SSTIC 2009, Rennes 2 juin 2009.